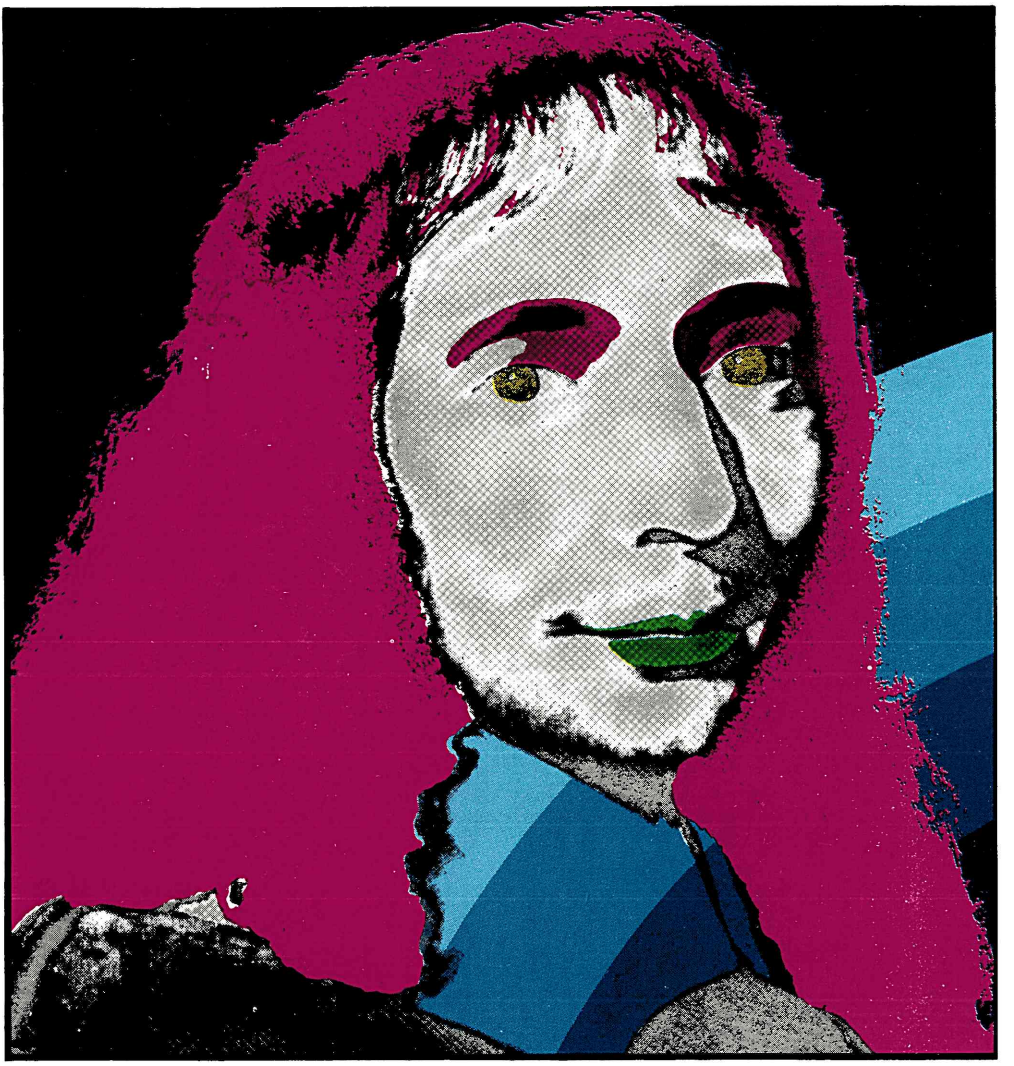
Turbo PASCAL 5.0

ARNE SCHÄPERS



KONZEPTE, ANALYSEN, TIPS & TRICKS

ADDISON-WESLEY

Turbo PASCAL 5.0

ARNE SCHÄPERS

ADDISON-WESLEY PUBLISHING COMPANY

Bonn · Reading, Massachusetts · Menlo Park, California · New York
Don Mills, Ontario · Wokingham, England · Amsterdam · Sydney · Singapore
Tokyo · Madrid · San Juan

CIP-Titelaufnahme der Deutschen Bibliothek

Schäpers, Arne
Turbo Pascal 5.0 : (Konzepte - Analysen - Tips - Tricks) / Arne Schäpers.- Bonn; Reading, Mass.; Menlo Park, Calif.; New York; Don Mills, Ontario; Wokingham, England; Amsterdam; Sydney; Singapore; Tokyo; Madrid; San Juan: Addison-Wesley,1989.
ISBN 3-89319-130-5

© 1989 Addison-Wesley (Deutschland) GmbH

Lektorat: Silke Lodes
Satz und Grafiken: Schäpers DaTec, München
Druck: Bercker Graph. Betrieb, Kevelaer
Umschlaggestaltung:Ulrich Carthaus, Alfter b. Bonn

Text, Abbildungen und Programme wurden mit größter Sorgfalt erarbeitet. Verlag, Übersetzer und Autoren können jedoch für eventuell verbliebene fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

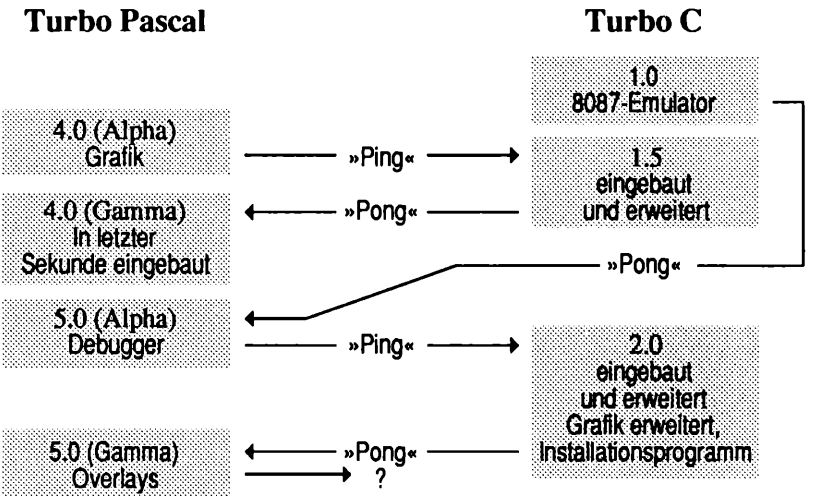
Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Kein Teil dieses Buches darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form durch Fotokopie, Mikrofilm oder andere Verfahren reproduziert oder in eine für Maschinen, insbesondere Datenverarbeitungsanlagen, verwendbare Sprache übertragen werden. Auch die Rechte der Wiedergabe durch Vortrag, Funk und Fernsehen sind vorbehalten.

Die in diesem Buch erwähnten Software- und Hardwarebezeichnungen sind in den meisten Fällen auch eingetragene Warenzeichen und unterliegen als solche den gesetzlichen Bestimmungen.

ISBN 3-89319-130-5

Ein Vorvorwort über die Evolution von Büchern und Programmen

Man stelle sich vor: da bringt Borland ein Pascal-Compilerpaket mit epochalen Neuerungen auf den Markt, präsentiert es als neuen Sprachstandard, und einer schreibt ein Buch darüber. Während derjenige an der Arbeit ist, gehen die Borland-Entwickler einem sehr kreativen Vergnügen nach - sie spielen Pingpong mit Ideen:



Was hier als Einzel dargestellt wird, ist in Wirklichkeit ein Mannschaftswettbewerb - unter anderem sind noch SideKick Plus, der Turbo Debugger, der Turbo Assembler und nicht zuletzt Turbo BASIC mit im Spiel.

Wo unser Autor geblieben ist? Er liegt mehr oder weniger unter dem Netz begraben und versucht dort, mit den deutschen Ausgaben der jeweiligen Dokumentationen sowie den »Updates« seiner eigenen Bücher nachzukommen*).

Turbo Pascal 5.0: Konzepte, Analysen, Tips & Tricks ist ein solches »Update« und basiert (wie auch die Compilerversion 5.0) zu 80 Prozent auf dem Material seines Vorläufers: zu bereits vorhandenen Teilen und Beschreibungen sind die Vergleiche

*) Soviel als Entschuldigung dafür, daß der bereits im Vorwort von *Turbo Pascal 4.0: Konzepte...* angekündigte zweite Band immer noch nicht erschienen ist. Natürlich haben solche Verzögerungen auch ihr Gutes: In diesem zweiten Band wird es jetzt (unter anderem) um die Möglichkeiten zur Nachkonstruktion von Overlays durch die Version 4.0 gehen. Wenn Sie daran interessiert sind, dann drücken Sie mir bitte die Daumen, daß die Schlußfolgerung des Diagramms - eine Overlay-fähige Version von Turbo C - nicht allzuschnell das Licht der Wirklichkeit erblickt...

zwischen den Versionen 4.0 und 5.0 und einige weitere Details hinzugekommen, für neue Konzepte (wie den 8087-Emulator) wurde die Struktur entsprechend erweitert.

Was Overlays betrifft, die in den folgenden Kapiteln nur recht stiefmütterlich davonkommen: der zweite Band (der nun wirklich in einigen Monaten erscheinen wird), widmet sich den technischen Hintergründen und Implikationen dieses Konzepts - nicht nur im Zusammenhang mit Speichererweiterungen - in aller Ausführlichkeit.

München, Dezember 1988 *Arne Schäpers*

Vorwort

Den »typischen« Leser*) stelle ich mir in einer Situation vor, die sich ungefähr so charakterisieren läßt:

- Das Benutzerhandbuch ist gelesen. Allzuviel an grundsätzlich Neuem war eigentlich nicht drin.
- Die interessanten Kapitel des Referenzhandbuchs (respektive: 23 und 25) sind teilweise Allgemeinplätze, teilweise ergehen sie sich in mehr oder minder dunklen Andeutungen und/oder offensichtlichen Unvollständigkeiten.
- Maschinensprache? `MOV AX,[0120]` bereitet wirklich keine Probleme - die fangen (wenn überhaupt) erst bei `LEA DI,ES:[BX+$FF00]` / `STOSB` an.
- Zeiger, Listen und Rekursion? Irgendwo zwischen einem vorsichtigen »Im Prinzip ja« und einem lauten »Nein danke«. (*Algorithmen und Datenstrukturen* von Niklaus Wirth ist auch bei mir nach einem kurzen Versuch für die nächsten Jahre wieder in den Bücherschrank gewandert).

Wer versucht, den Abläufen mit DEBUG auf die Spur zu kommen, sieht sich dem üblichen Chaos von Maschinenbefehlen gegenüber, das ein Hochsprachen-Compiler nun einmal erzeugt. Dazu kommt eine Unzahl von Aufrufen der Laufzeitbibliothek: *System* besteht aus mehreren Kilobytes hochkonzentriertem Code, der dem vorwitzigen Forscher normalerweise den Rest gibt.

Ich hatte mir eine Art Fortsetzung der offiziellen Dokumentation vorgenommen - ein Buch, das

- alles enthält, was in den offiziellen Handbüchern von Turbo Pascal fehlt oder wofür einfach kein Platz mehr war (inklusive eines *echten* Vergleichs mit der Version 3.0),
- verschiedene Unerfreulichkeiten umschifft (hauptsächlich: die fehlende Unterstützung deutscher Umlaute in Grafik-Zeichensätzen),
- einiges an »Utilities« bietet, die man eigentlich schon immer gebraucht hätte (Debugger für den 8087, Speichern von DEBUG-Listings in Dateien, Directory-Absuche über Datum und Uhrzeit etc.),
- und schließlich eine Art Brücke schlägt zwischen Pascal-Programmierern, Maschinenfreaks und Universitätsprofessoren.

So etwas ähnliches ist *Turbo Pascal: Konzepte, Analysen Tips & Tricks* schließlich auch geworden: definitiv kein Material für Anfänger, sondern ein bunter Streifzug durch Hintergründe und Möglichkeiten.

---

*) Der/die Leser(in) möge es verzeihen, wenn er/sie keine gesonderte Anrede erfährt - schließlich soll man/frau bzw. der/die Programmierer(in) die(se(n)) S(a/ä)tz(e) ja noch lesen können.

»Vollständig« ist dieses Buch allerdings bei weitem nicht - die Möglichkeiten und die Komplexität der Versionen 4.0 und 5.0 sind so überwältigend, daß auf den folgenden fast 500 Seiten neben grundlegenden Strukturen gerade noch die drei Schwerpunkte »TSR-Programme«, »Grafik« und »Gerätetreiber« behandelt werden konnten.

Aus diesem Grund wird in den nächsten Monaten ein zweiter Band erscheinen, der sich unter anderem auf serielle Kommunikation, Netzwerke und Speichererweiterungen konzentriert.

Noch ein Nachsatz zum Thema »Urheberrecht«: Mir stehen über Turbo Pascal genau dieselben Informationen zur Verfügung wie jedem anderen Benutzer auch. Sämtliche Analysen der verschiedenen Versionen, Listings, Behauptungen etc. der folgenden Kapitel sind deshalb genau das, was ihre Bezeichnungen implizieren: eine *Interpretation.*

Anders gesagt: Ich habe in die offizielle Dokumentation zu Turbo Pascal alles hineingepackt, was von jenseits des großen Teichs nach Deutschland gekommen ist - wenn ich von Borland irgendwelche weiteren »internen« Informationen erhalten hätte, wäre dieses Buch ein Verstoß gegen geltende Gesetze und/oder Verträge mit der Firma Heimsoeth (die in diesem Sinne - mangels Masse - nicht existieren).

München, Januar 1988 *Arne Schäpers*

Ein Hinweis des Verlages: Die im Index verzeichneten Beispielprogramme sind zusammen mit erweiterten Grafik-Zeichensätzen auf Diskette für DM 91.20 (inklusive MwSt.) über den Autor erhältlich. Bestellungen bitten wir zusammen mit der Angabe des benötigten Diskettenformats (3,5 oder 5,25 Zoll) schriftlich an die folgende Adresse zu richten:

Schäpers DaTec
Postamt Ottobrunn
postlagernd
8012 Ottobrunn

Inhaltsverzeichnis

Kapitel 1
Speicherstrukturen 1
Die Organisation des Hauptspeichers 1
Ein Programm im Speicher 4
Environment-Tabelle und Programmsegment-Präfix 4
HiMem 5
Environ 6
CmdLine 7
Segmente eines Programms 8
Ein Einschub zum Thema »Relozieren« 12
Ein »typisches« Pascal-Programm 13
Units, Variablen und Konstanten 18
Die Initialisierungsteile der Standard-Units 21
Der Initialisierungsteil des Units *System* 22
Das Unit *Overlay* nebst seiner Initialisierung 31
Der Initialisierungsteil von *Crt* bzw. *Dos* 33
Der Initialisierungsteil von *Printer* 38
Der Initialisierungsteil von *Graph* 39
Der Initialisierungsteil von *Turbo3* bzw. *Graph3* 42
Typisierte und untypisierte Konstanten 43
Prozeduren und Funktionen 50
Parameterlose Funktionen 50
Stringfunktionen 56
Realzahlen als Funktionsergebnisse 62
Parameter und lokale Variablen 62
Einfache Wertparameter bei Prozeduren 63
Variablenparameter 66
Strukturierte Wertparameter 70
Unterschiede zwischen Funktionen und Prozeduren 74
Vorsicht mit **inline**! 77
inline mit symbolischen Namen - eine zweischneidige Lösung 80
Genauso gefährlich: numerische **const**-Deklarationen 83
inline-Deklarationen 83
Prozeduren und Funktionen in Maschinensprache 86
Eine deutsche Version von *UpperCase* 89
Pseudo-**var**-Parameter und die Folgen 92

Kapitel 2
Der Coprozessor 8087 93
Zum Thema »Parallelverarbeitung« 95
Die Register des 8087 97

Das »Tag«-Register 100
Das Statuswort-Register 102
Das Kontrollwort-Register 104
Die Initialisierung des 8087 105
... durch Turbo Pascal in der Version 4.0 105
Ein Einschub zum Prinzip des 5.0-Emulators 108
Die beiden Versionen von INIT87 111
Die Prüfung des Systems 113
Realzahlen als Parameter 119
Mehrere und »gemischte« Parameter 123
Funktionen mit Fließkomma-Ergebnistypen 127
Variablenparameter 130
Real-Variablen - ein schreckliches Thema 131
Ein vollständiges Beispiel 132
Real-Variablen im Modus {$N+} 138
Geschwindigkeit und Präzision 141
Für Benchmark-Tester 146
Direkte Programmierung 148
Apfelmännchen mit Köpfchen 149
Die Pascal-Version 150
Der Ersatz von *Iterate* durch eine Maschinenroutine 153
Der Ersatz des Kreis-/Zykloidentests durch eine Maschinenroutine 157

Kapitel 3
Zeiger und Rekursion 163
Zeigervariablen 163
Variablengrößen und -strukturen 164
Generische Zeigertypen 168
Zum Thema »Notation« 168
Explizite Typ-Umwandlungen 172
Zeiger- vs. Wertkonvertierungen und Probleme mit Arrays 173
Zugriffe auf Teile einer Struktur 175
»Statische« Anwendungen 179
StrSort1 - die Standardvariante 179
StrSort2 - dasselbe Programm mit Zeigern 181
»Dynamische« Variablen 183
Zeiger auf Pascal-Routinen in der Version 4.0 185
PROCFUNCPTR.PAS 186
PROCARRAY.PAS - Aufruf über Funktionsnummern 187
In der Version 5.0...
Rekursion 191
Noch einmal kurz zum Thema »Stack und Lokales« 191
Regeln, Forderungen und Vorsichtsmaßnahmen 193

Absuche von Directorystrukturen 195
Rekursion und Schleifen kombiniert 199
Die Ausgabe eines Dateieintrags: DSPENTRY.INC 204
DATEFIND 205
Lineare dynamische Listen 211
Strings und die Prozedur *GetMem* 212
Einfache Listen 213
Einfügen und Löschen von Elementen 214
Doppelt verkettete Listen 221
Einfügen und Löschen 222
»Dummy«-Elemente 223
LST2DEMO 224

Kapitel 4
Speicherresidente Programme und Interrupts 229
Die Funktion KEEP 230
interrupt-Prozeduren 233
»Einhängen« zusätzlicher Funktionen 235
interrupt-Deklaration := Aprilscherz 236
INTFRAME.ASM 237
Ein Demonstrationsprogramm zu INTFRAME 241
Eine echte Anwendung: TRACE86 242
Hilfe für den Zauberlehrling: Entfernbare TSR-Programme 245
Funktionsgesteuerte Software-Interrupts 246
Hardware-Interrupts oder: Die Allgemeine Lösung 251
Vorsicht mit TURBO-eigenen Vektoren! 259
Entfernte TSR-Programme, Speicherlöcher und Interruptketten 260
Mehrere Programme über einen Interrupt-Vektor 261
DOS über die Schulter geschaut 262
Zyklischer Aufruf über DOS-Funktionen 264
Ein Demonstrationsprogramm zu DOSHOOK 267
Abfangen von BIOS-Funktionen über DOSHOOK 270
Licht ins Dunkel des Coprozessors: TRACE287 279
Ersatz von DOS-Funktionen 288
CATCH22 291
TPCED - ein Kommandozeilen-Editor 294
TPCED0.PAS - die »Trockenversion« 296
TPCED als speicherresidentes Programm 302
Eine neue Version von *ReadKey* 306
Zur Organisation des Tastaturpuffers 307
Der Interrupt $1B 308
TPCED mit Erweiterungen 309

Kapitel 5
Grafik 315
Zur Struktur von *Graph* 315
Ein Aufruf von *PutPixel* 316
Die Lage von *Graph* und Grafiktreibern im Speicher 320
Dynamische Belegung über *InitGraph* 321
Statische Belegung - Grafiktreiber als .OBJ-Dateien 324
Die Initialisierung 326
Grafiktreiber im Detail 327
EGAVGA.BGI - die Startadressen der einzelnen Funktionen 330
Bildformate und Parametertabellen 333
Eigene Bildformate für EGA-Karten mit *Graph* 334
Spezifische Initialisierungen 334
640 * 480 Bildpunkte für die GENOA HIRES 334
SetGraphMode aus der Sicht des Grafiktreibers 335
Eigene Initialisierungen 340
Beliebige Bildformate 341
Grafik-Zeichensätze 343
Vektordefinitionen 345
Ausgabe von Zeichendefinitionen mit einem Pascal-Programm 348
Die Erweiterung von Zeichensätzen 352
Anfügen und Löschen von Zeichendefinitionen 354
»WYSIWYG« 355
CE - Der Editor für grafische Zeichensätze 356
Der Start von CE 357
Die Auswahl von Menüpunkten 358
Das Hauptmenü 358
Das Untermenü »Zeichen« 358
Der Editor 359
Die Programmstruktur von CE 361
Das Unit *CEGlobal* 361
Das Unit *CEUtils* 363
Das Unit *CEEdit* 375
Das Hauptprogramm: CE.PAS 386
Lesen anderer Zeichensätze: Die Prozedur *ImportChar* 396
Neue Versionen von *OutText* 400
Laden und Speichern von Grafikbildern 404

Kapitel 6
Dateivariablen und Gerätetreiber 409
Die Evolution von Dateivariablen 409
Eine Erweiterung durch die Version 3.0 409
... und eine zweite durch die Version 4.0 410
»Treiber«-Routinen müssen... 411

Druckerausgaben im Eiltempo 412
Umleitungsmöglichkeiten für Ein- und Ausgaben 416
Ausgabeketten 419
AssignCrt und eigene Routinen 420
Der Ersatz von *ConInPtr* und *ConOutPtr* in 3.0-Programmen 420

Anhang A
Programmiermodell des 8086 425
Die Register des 8086 425
Was ist ein Register? 426
Allzweck-Register 427
Segmentregister 428
Spezialregister 428
Der Stack und Unterprogramme 429
Segmentierung des Speichers 434
Aufrufe als near und far 438
Interrupts und Vektoren 439

Anhang B
Einige Hinweise zu MASM 443
Der Aufruf von MASM 443
Programmcode 444
Deklarationen und near-Labels 446
EXTRN-Deklarationen 446
Datendeklarationen 447
Datengrößen 449
Explizite Größenbestimmung 450
Daten vs. Adressen 451
forwards jeder Art 451

Anhang C
Der 8087-Emulator im Detail 453
8087-Opcodes und INT $3x-Befehle 453
Der Dispatcher für INT $34..$3D 455
Der »Shortcut«-Interrupt $3E 459
Der 8086-Emulator bzw. eine Entschuldigung 464

Anhang D
Zusammenfassung, Nachgedanken & Tips 467
Explizite Typ-Umwandlung von Zeigern 470
Mehrere und simultane Interrupts mit einem TSR-Programm 472
Allgemeines 475
Formatierte Ausgabe von Nullstrings 475
Start- und Endwerte von **for**-Schleifen 475

MaxColors 475
Compilerfehler 476
Sprungadressen im Datensegment 476
Adressen von Pascal-Routinen im Codesegment 477

Index 481

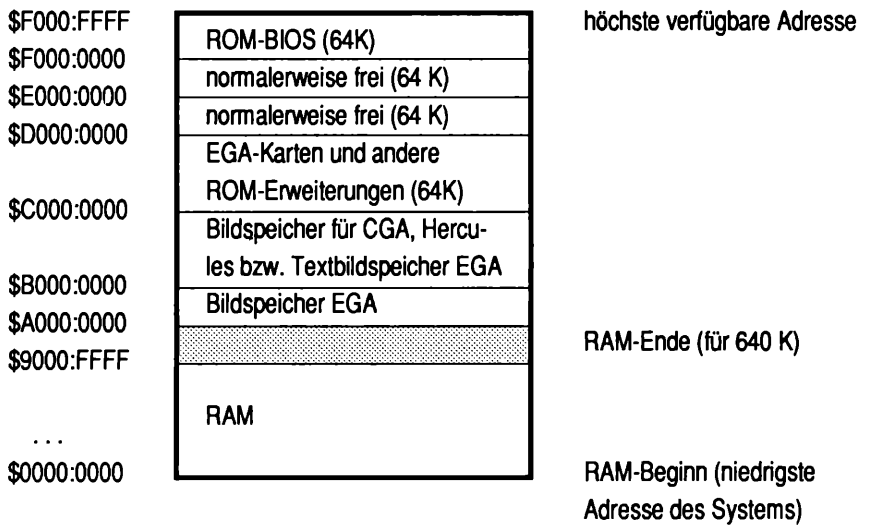
Kapitel 1
Speicherstrukturen

»Der Adressraum des Prozessors ist durch die Architektur des Systems in RAM- und ROM-Bereiche gegliedert, wobei der RAM vom Betriebssystem in Speicherblöcke zur Aufnahme von Programmen unterteilt wird. Ein Programm besteht aus einzelnen Segmenten, von denen einige Prozeduren und Funktionen enthalten, die ihrerseits weitere Routinen lokal definieren können«. Fast fühlt man sich an ein Gemälde von Escher (oder auch zeitgeistiger: an Apfelmännchen) erinnert.

Dieses Kapitel stellt einen Versuch dar, die ineinander verschachtelten Strukturen schrittweise zu entwirren. Trotz des unbestreitbaren Vergnügens an Mikro- und Makrokosmos endet es allerdings mit Prozessorbefehlen und läßt den internen Microcode des 8086 außer acht, der seinerseits für jeden Befehl ein Programm enthält, das aus einzelnen Routinen besteht, die ihrerseits weitere Routinen...

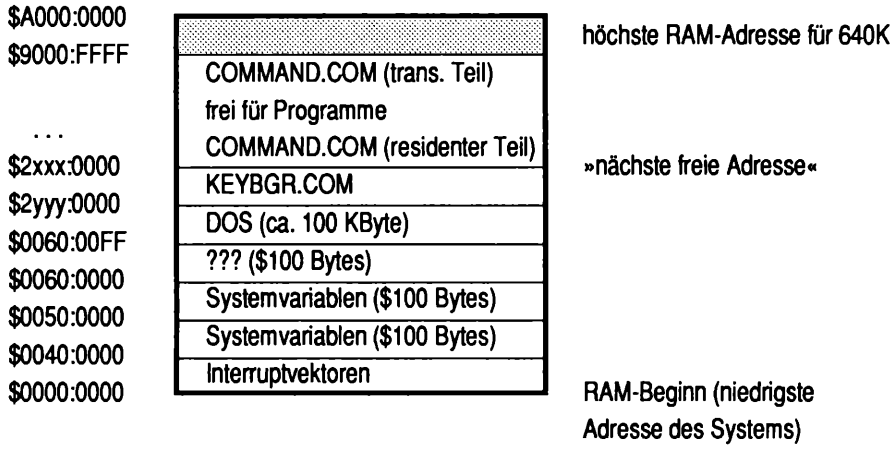
Die Organisation des Hauptspeichers

Der 8086 und seine Nachfolger haben einen Adressbereich von einem Megabyte, d.h. können insgesamt 1048576 verschiedene Speicherzellen ansprechen. (Die Umschaltungsmöglichkeiten in den »protected mode« ab dem 80286 sowie I/O-Adressen sind in dieser leicht vereinfachten Betrachtung nicht berücksichtigt). In den Computern der PC-Serie ist dieser Adressraum folgendermaßen aufgeteilt:



Ein kurzer Einschub: Beim Start des Computers (oder einem Druck auf den RESET-Knopf) führt der Prozessor einen Sprung zur Adresse $FFFF:0000 (anders notiert: $F000:FFF0) aus. Der erste Befehl, der an dieser Stelle aus dem ROM gelesen wird, besteht normalerweise aus einem Sprung zu einer ROM-Routine, die den RAM-Test ausführt, grundlegende Initialisierungen des Systems vornimmt und schließlich den »Boot«-Sektor einer eingelegten Diskette bzw. der Festplatte liest. Mit den aus diesem Sektor gelesenen Befehlen geht es dann weiter.

Wir konzentrieren uns auf den RAM-Bereich, der nach dem Start des Systems in etwa so aufgeteilt ist:



Der Bereich von $0000:0 bis $0000:03FF (anders notiert: $0030:00FF) ist für 256 Interruptvektoren reserviert, von denen jeder 4 Bytes belegt. Ein Teil dieser Vektoren wird durch die Initialisierungsroutine des ROMs gesetzt, ein zweiter bei der Initialisierung von DOS. Kapitel 4 beschäftigt sich eingehend mit den Anwendungsmöglichkeiten von Interrupts, Anhang A enthält einen kurzen Abriß der zugrundeliegenden Mechanismen.

Der Bereich von $0000:0400 bis $0000:06FF (anders notiert: $0040:0 ... $0060:00FF) wird für eine Reihe von »Systemvariablen« verwendet - hier finden sich die physikalische Größe des RAMs ($40:13/14), die Statusbits der Tastatur ($40:17/18), der Tastaturpuffer, der Status der Diskettenlaufwerke, der momentan gesetzte Videomodus usw. Die höchste Speicheradresse in diesem Bereich, die Peter Norton in seinem *Programmer's Guide* auflistet, hat die Adresse $0000:051A - die darüberliegenden rund 400 Bytes sind laut *DOS Technical Reference Manual* für interne Zwecke von DOS reserviert.

Das Codesegment des ersten Teils von DOS (IBMBIO.COM) beginnt unabhängig von der DOS-Version mit dem Segment $0070, IBMBIO.COM endet irgendwo innerhalb des Bereichs $1000:xxxx. Direkt daran schließt sich der zweite Teil (IBMDOS.COM) an, dessen Endadresse nicht nur von der jeweiligen Version,

sondern auch von der Anzahl gesetzter Dateipuffer, zusätzlich geladener Gerätetreiber usw. abhängig ist - sie liegt meist irgendwo im Bereich $22xx:0000.

KEYBGR wurde deshalb mit in diese Betrachtung aufgenommen, weil es hierzulande praktisch als Bestandteil des Betriebssystems angesehen werden kann. Dieses Programm wird - einen entsprechenden Befehl in AUTOEXEC.BAT vorausgesetzt - ab der Endadresse von DOS in den Speicher geladen und nach seinem Ablauf nicht wieder entfernt (siehe Kapitel 4). Die »nächste freie Adresse« verschiebt sich durch dieses »Einlagern« entsprechend nach oben.

Ab der »nächsten freien Adresse« steht der *transiente* Teil des Programms COMMAND.COM im Speicher - jedenfalls, solange wir uns auf der Kommandoebene von DOS befinden. Tatsächlich besteht COMMAND.COM aus zwei Teilen:

- Einem *residenten* Teil, der beim Start von DOS auf ähnliche Art wie KEYBGR in den Speicher geladen wird. Dieser Teil befindet sich *unterhalb* von KEYBGR - zu ihm führt der »Rücksprung zu DOS« beim Beenden eines laufenden Programms. (Hier findet sich übrigens auch der Code für die Ausgabe der berüchtigten Meldung `Abort Retry Ignore` bei Diskettenfehlern).
- Einem *transienten* Teil, der Funktionen wie `dir`, `del` usw. enthält. Dieser Teil wird oberhalb von KEYBGR - auf die »nächste freie Adresse« - geladen und kopiert sich von dort aus an das obere Ende des Hauptspeichers (auf Systemen mit 640 KByte RAM etwa in den Bereich $9000:B800..FFFF).

Der residente Teil von COMMAND.COM ermittelt nach jedem »Rücksprung zu DOS« über eine Prüfsumme, ob die Anwendung den transienten Teil überschrieben hat - falls ja, wird das gesamte Programm COMMAND.COM neu von der Diskette geladen und der transiente Teil wieder ans obere Ende des Hauptspeichers kopiert*).

In den folgenden Abschnitten gehen wir auf diese Zweiteilung nur noch einmal ein - abgesehen von etwas kürzeren »Nachladezeiten« bei der Arbeit mit Diskettenlaufwerken hat sie im praktischen Betrieb keine weitere Bedeutung. Oder auch: Wenn sich die Datei COMMAND.COM nach der Ausführung einer Anwendung nicht finden oder laden läßt, muß das System neu gestartet werden, weil der residente Teil nicht allein lauffähig ist.

*) Der residente Teil bleibt in jedem Fall unverändert - die entsprechenden Daten werden bei einem Neuladen von COMMAND.COM zwar ebenfalls von der Diskette gelesen, ansonsten aber ignoriert.

Ein Programm im Speicher

Wenn NUMMER1.EXE ein ausführbares Programm darstellt, dann läßt sich die Ausführung des Befehls `nummer1` auf der Kommandoebene von DOS in drei Schritte unterteilen:

1. COMMAND.COM stellt fest, daß es sich nicht um ein eingebautes Kommando (wie `dir` oder `del`) handelt, interpretiert die eingegebenen Zeichen deshalb als Dateinamen und lokalisiert NUMMER1.EXE über entsprechende DOS-Aufrufe auf der Diskette/Festplatte.
2. Über den Aufruf der DOS-Funktion EXEC wird NUMMER1.EXE ab der »nächsten freien Adresse« in den Speicher geladen und gestartet. (Die »Ladekopie« des transienten Teils von COMMAND.COM wird bei dieser Aktion ganz oder teilweise überschrieben).
3. Das Programm wird ausgeführt und endet mit dem DOS-Aufruf »Terminate«. Dieser Aufruf führt letztendlich in den residenten Teil von COMMAND.COM hinein, der nötigenfalls den transienten Teil mit entsprechenden DOS-Aufrufen wieder von der Diskette lädt (und in diesem Fall den zuvor von NUMMER1.EXE belegten Speicherbereich überschreibt).

Ein **Hinweis:** TSR-Programme und Löcher in dem von DOS belegten Speicherplatz sind hier nicht berücksichtigt - siehe Kapitel 4.

Environment-Tabelle und Programmsegment-Präfix

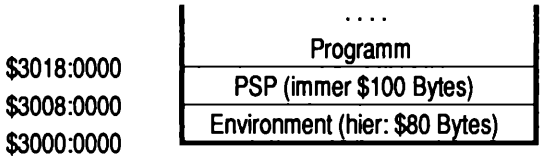
DOS stellt einem Programm beim Laden zwei Speicherbereiche mit Informationen voran, die jeweils eine eigene Segmentadresse haben:

- das Programmsegment-Präfix (PSP) mit einem Umfang von 256 Bytes;
- eine Kopie der *Environment*-Tabelle mit unbestimmter Länge, die jeweils auf eine Segmentgrenze aufgerundet wird, d.h. auf ein ganzzahliges Vielfaches von (dezimal) 16.

Diese beiden Speicherbereiche werden *jedem* Programm vorangestellt - unabhängig, ob es mit Turbo Pascal oder auf andere Weise erzeugt wurde.

Die Segmentadresse des PSP wird dem Programm im Register ES übergeben und von Turbo Pascal in der Variablen *PrefixSeg* gespeichert; die Segmentadresse der Environment-Tabelle ist in einem Feld des PSP enthalten.

Wenn wir annehmen, daß der transiente Teil von COMMAND.COM ab der Adresse $3000:0000 im Speicher steht, dann ergibt sich ungefähr die folgende Verteilung für das Laden eines beliebigen Programms:



Die wichtigsten Teile des Programmsegment-Präfix als Pascal-Datentyp:

```
type
  PSPType = record
    Fill1:   array[0..1] of Byte;
    HiMem:   Word;                  { <$02> RAM-Endsegment }
    Fill2:   array[0..39] of Byte;
    Environ: Word;                  { <$2C> Environment-Segmentadresse }
    Fill3:   array[0..81] of Byte;
    CmdLine: string[127];           { <$80> Kommandozeilen-Parameter }
  end;                              { mit vorangestelltem Längenbyte }
```

HiMem

Der Wert dieses Feldes ist das Ergebnis einer Kette von Additionen beim Laden der Datei:

```
  Startadresse im Speicher (= nächste freie Adresse + Environment + PSP)
+ Umfang des Programms     (= Codesegmente + Datensegment + Stack)
+ HeapMax
```

Die Startadresse des Programms wird von DOS eingesetzt, die restlichen Daten hat der Linker in der .EXE-Datei verzeichnet.

Wenn das Ergebnis dieser Addition die Endadresse des tatsächlich vorhandenen Hauptspeichers überschreitet, findet eine automatische Begrenzung statt - ein Turbo Pascal-Programm, das mit der Standardvorgabe *HeapMax* = 655360 arbeitet, bekommt in *HiMem* also immer die tatsächliche Obergrenze des physikalischen Hauptspeichers übergeben ($A000 bei 640 KByte RAM).

Wenn *HeapMax* mit einem Compilerbefehl auf einen niedrigeren Wert gesetzt ist, dann trägt DOS das tatsächliche Ergebnis der Berechnung in *HiMem* ein: Für ein Programm mit $1000 Paragraphs Umfang (inklusive Environment und PSP), das mit *HeapMax* = $2000 übersetzt wurde und von DOS ab der Adresse $3000 in den Speicher geladen wird, hat *HiMem* den Wert $6000.

Im Rahmen einer eventuellen Begrenzung von *HiMem* auf die Endadresse des physikalischen Hauptspeichers wird auch die Prüfung mit *HeapMin* ausgeführt: Wenn das Ergebnis der Berechnung »Startadresse + Programmumfang + HeapMin« die Obergrenze des physikalischen Speichers überschreitet, bricht DOS mit einer entsprechenden Fehlermeldung ab. (Diese Prüfung findet also außerhalb des Pascal-Programms statt).

Das Programm bekommt das Ergebnis der Berechnungen mit *HeapMin* und *HeapMax* nur in der Form von *HiMem* übermittelt und verwendet diesen Wert bei der Initialisierung von *FreePtr* (siehe »Der Initialisierungsteil des Units *System*«).

Environ

Dieses Feld enthält die Segmentadresse einer Kopie der momentan gültigen *Environment*-Tabelle, die sich normalerweise direkt unterhalb des PSP im Speicher befindet. Die Tabelle selbst besteht aus einer ungeordneten Folge von ASCIIZ-Strings (d.h. Zeichenfolgen ohne vorangehendes Längenbyte, die jeweils mit dem Zeichen `chr(0)` (= NUL) abgeschlossen sind); das Tabellenende wird durch zwei aufeinanderfolgende NUL-Zeichen markiert. Erstaunlicherweise haben die Entwickler von Turbo Pascal erst ab der Version 5.0 ein Gegenstück zu *ParamStr* und *ParamCount* für *Environment*-Einträge vorgesehen - »4.0-Programmierer« müssen sich deshalb mit Eigenkonstruktionen helfen, die in etwa so aussehen könnten wie das folgende Listing:

```
program EnvironmentDemo;   { ENVP.PAS, 28/12/87, 29/11/88 as }
{ 4.0-Ersatz für die Funktionen EnvStr und EnvCount }

function EnvStr(Number: Integer): string;
{ Liefert den x. Eintrag der Tabelle Environment als Pascal-String }
type
  ChArray = array[0..2000] of Char;
var
  p: ^ChArray;            { Zeiger auf ein Char-Array }
  x, ELength: Integer;    { Index, Längenzähler }
begin
  if Number <= 0 then
  begin
    EnvStr := '';   { EnvStr(0) ist nicht definiert }
    Exit;
  end;
  p:= Ptr(Word(Ptr(PrefixSeg,$2C)^),0);  { p auf den Anfang }
  x:= 0;
  while (Number -1 > 0) and (p^[x] <> #0) do
  begin
    Dec(Number);
    while p^[x] <> #0 do Inc(x); { Erhöhen bis zum Eintragsende }
    Inc(x);   { #0 am Ende des Eintrags überspringen }
  end;
  { p zeigt jetzt auf den Anfang des gewünschten Eintrags }
  ELength := 0;  { Längenzähler }
  while p^[x] <> #0 do
  begin
    EnvStr[ELength+1] := p^[x];  { Kopie des Eintrags }
    Inc(ELength);
    Inc(x);
  end;
  EnvStr[0] := chr(ELength);  { Längenbyte }
end;
```

```
function EnvCount: Integer;
{ Liefert die Anzahl der Einträge in der Environment-Tabelle zurück }
var x: Integer;
begin
  x:= 1;
  while Length(EnvStr(x)) <> 0 do Inc(x);
  EnvCount := x-1;
end;

var x: Integer;

begin
  Writeln(EnvCount, ' Einträge der Tabelle Environment:');
  for x:= 1 to EnvCount do
    Writeln('  ',EnvStr(x));
end.
```

Eine typische Ausgabe dieses Programms könnte so aussehen:

```
3 Einträge der Tabelle Environment:
  COMSPEC=C:\COMMAND.COM
  PATH=\DOS;\TP40;\NU;\WORD
  PROMPT=$P$G
```

CmdLine

Dieses Feld des PSP enthält die Kommandozeilen-Parameter beim Aufruf des Programms in Form eines Pascal-Strings, d.h. mit einem vorangestellten Längenbyte. Der Name des Programms ist nicht enthalten - der String beginnt statt dessen mit dem ersten Zeichen, das DOS als Trennung zwischen dem Programmnamen und dem ersten Parameter erkannt hat. Ein Beispiel:

```
program CmdLine;
var p:^string;

begin
  p := Ptr(PrefixSeg,$80);   { p auf die Kommandozeilen-Parameter }
  Writeln('Parameter:',p^);
end.
```

Der Aufruf dieses Programms mit `CmdLine Parm1` ergibt `Parameter: Parm1`, der Aufruf mit `CmdLine/x` ergibt `Parameter:/x` - im zweiten Fall erkennt DOS den Schrägstrich vor dem `x` als Trennzeichen.

Die Standardfunktionen *ParamStr* und *ParamCount* greifen direkt auf *PSP.CmdLine* zu, wobei - im Gegensatz zu COMMAND.COM - nur Leerzeichen als Trennung zwischen zwei Parametern behandelt werden.

Segmente eines Programms

Zur Demonstration definieren wir ein Programm, das nur aus **begin** und **end** besteht:

```
program Null;   { NULL.PAS }
{$IFDEF TP40} {$T+   .TPM-Datei erzeugen, für 5.0 nicht definiert} {$ENDIF}
begin
end.
```

Die Compilierung dieses Programms als .EXE-Datei und der nachfolgende Aufruf von TPMAP ergeben eine .MAP-Datei, in der die folgenden Segmente verzeichnet sind:

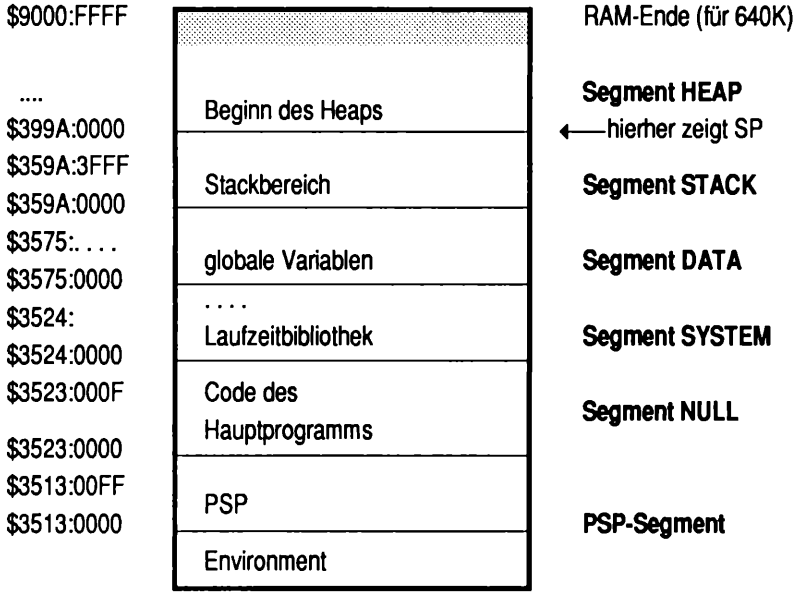
```
Start  Stop   Length Name              Class

00000H 0000FH 00010H NULL              CODE
00010H 00511H 00502H SYSTEM            CODE
00520H 00768H 00249H DATA              DATA
00770H 0476FH 04000H STACK             STACK
04770H 04770H 00000H HEAP              HEAP
```

Das Codesegment des Units *System* beginnt $10 Bytes (= 1 »Paragraph«) hinter dem Codesegment des Programms. $520 Bytes (= $52 Paragraphs) oberhalb des Programm-Codesegments beginnt das Datensegment; das Segment für den Stack beginnt $77 Paragraphs oberhalb des Programmstarts. Mit DEBUG oder SYMDEB läßt sich zeigen, daß diese »Abstände« der einzelnen Segmente auch beim Laden des Programms konstant bleiben:

```
debug null.exe
-u 0,b
3523:0000 9A00002435    CALL    3524:0000       ; SYSTEM: Initialisierung
3523:0005 89E5          MOV     BP,SP           ; "begin"
3523:0007 89EC          MOV     SP,BP           ; "end"
3523:0009 31C0          XOR     AX,AX           ; Exitcode 0
3523:000B 9AF3012435    CALL    3524:01F3       ; Ende über Aufruf von HALT
-r
AX=0000  BX=0000  CX=0350  DX=0000  SP=4000  BP=0000  SI=0000  DI=0000
DS=3513  ES=3513  SS=359A  CS=3523  IP=0000   NV UP EI PL NZ NA PO NC
```

Das Hauptprogramm beginnt mit dem Aufruf des Initialisierungsteils von *System* und endet mit einem Aufruf der Prozedur *Halt*, die ebenfalls in *System* gespeichert ist.

Die absoluten Segmentadressen spielen keine Rolle (und werden aller Wahrscheinlichkeit nach auf Ihrem Computer abweichen) - entscheidend ist, daß sich *System* im gezeigten Listing eine »Segmentadresse« (also $10 Bytes) oberhalb des Programmsegments befindet. Derselbe Zusammenhang zwischen den in der .MAP-Datei verzeichneten Programmteilen und absoluten Segmentadressen gilt auch für DATA, STACK und HEAP:



Wie der r-Befehl von DEBUG zeigt, sind durch das Laden von NULL.EXE auch einige Register des Prozessors gesetzt worden:

- CS:IP enthalten die Adresse des ersten Programmbefehls ($3523:0000).
- SS:SP zeigen auf das obere Ende des Stacks: Der Wert $4000 von SP entspricht der Standardvorgabe {$M 16384, ...}. (SP wird *vor* Speicherungen auf dem Stack erniedrigt, der Aufruf von *System* am Anfang des Hauptprogramms schreibt die Rücksprungadresse also in den Bereich SS:$3FFC..3FFF).
- DS und ES enthalten die Segmentadresse des PSP ($3513).

Wenn wir NULL.EXE durch eine Umbenennung als »Nicht-.EXE-Datei« tarnen, läßt sich zeigen, wo diese Registerwerte herkommen:

```
ren null.exe null
debug null

-d 100,11b
34F9:0100  4D 5A 50 01 03 00 04 00-03 00 25 04 25 A4 77 00
34F9:0110  00 40 00 00 00 00 00 00-1C 00 00 00

-d 11c,12f
34F9:012F                                       03 00 00 00
34F9:0120  0E 00 00 00 01 00 01 00-F8 01 01 00 00 00 00 00
```

```
-u130,13f
34F9:0130 9A00000100    CALL    0001:0000
34F9:0135 89E5          MOV     BP,SP
34F9:0137 89EC          MOV     SP,BP
34F9:0139 31C0          XOR     AX,AX
34F9:013B 9AF3010100    CALL    0001:01F3
```

Hier haben wir die .EXE-Datei in der Form, in der sie auf der Diskette gespeichert ist. Vor dem Code des Programms befindet sich eine *Kopfinformation*, die im Normalfall von DOS beim Laden ausgewertet wird. Sie setzt sich aus zwei Teilen zusammen: einer Relozierungstabelle, die in unserem Beispiel mit dem Byte $1C der Datei (DEBUG-Adresse $011C) beginnt, und dem eigentlichen Dateikopf, der die Bytes $00..$1B der Datei belegt. Das *DOS Technical Reference Manual* gibt Auskunft über Struktur und Verwendungszweck dieser Daten:

Tabelle 1.1 - Kopfinformation einer .EXE-Datei

Byte(s)	Bedeutung
$00/01	$4D, $5A - Signatur des Linkers. An diesen beiden Bytes erkennt DOS, daß es sich um eine »gültige« .EXE-Datei handelt.
$02/03	Länge der Datei modulo 512
$04/05	Länge der Datei in Einheiten zu je 512 Bytes (inklusive der Kopfinformationen)
$06/07	Anzahl der Einträge in der Relozierungstabelle
$08/09	Größe der Kopfinformationen (inklusive der Relozierungstabelle) in Paragraphs (Einheiten zu je 16 Bytes)
$0A/0B	*HeapMin* (in Paragraphs)
$0C/0D	*HeapMax* (in Paragraphs)
$0E/0F	Abstand des Stacksegments zum ersten Codesegment des Programms in Paragraphs
$10/11	Anfangswert von SP (im Stacksegment)
$12/13	Prüfsumme für die gesamte Datei
$14/15	Anfangswert von IP (im ersten Codesegment)
$16/17	Adresse des ersten Codesegments in der Datei (in Paragraphs)
$18/19	Adresse des ersten zu relozierenden Elements innerhalb des ersten Codesegments (in Bytes)
$1A/$1B	Overlay-Nummer (0 für das Hauptmodul eines Programms)

Aus drei dieser Daten (Abstand des Stacksegments, Startwert für SP und Startwert für IP) lassen sich einige Folgerungen ableiten:

- sowohl die Größe des Stacks als auch seine Position relativ zum Programmbeginn ist festgelegt;
- der erste Befehl des Programms muß innerhalb des ersten Codesegments stehen (einen relativen Startwert für CS gibt es nicht). Es ist allerdings nicht notwendig, daß dieser Befehl auf der Adresse 0000 dieses Codesegments steht (tatsächlich ist das bei einem Turbo Pascal-Programm auch nur dann der Fall, wenn es keine Prozeduren und Funktionen definiert).

Direkte Informationen über das Datensegment (oder theoretisch: die Datensegmente) eines Programms gibt es im Kopf einer .EXE-Datei nicht. Das Datensegment könnte sich also oberhalb oder unterhalb des Stacksegments im Speicher befinden.

Wie bereits vorweggenommen, verwenden Turbo Pascal-Programme die zweite Möglichkeit, d.h. definieren das Datensegment unterhalb des Stacks und laden das DS-Register im Initialisierungsteil von *System* mit der Konstanten DATA. Diese Konstante ist innerhalb der .EXE-Datei als »relocatable« gekennzeichnet, wird also beim Laden an die tatsächliche Adresse des Datensegments angepaßt.

Version 5.0

Wenn man davon absieht, daß der Compiler-Schalter {$T} nicht definiert ist und die Erzeugung einer .MAP-Datei nach Setzen des entsprechenden Menüpunkts (*O/L/Map file/Detailed*) ohne den Umweg über das Programm TPMAP geschieht, beschränken sich die Unterschiede auf Adressen:

Zum einen enthält das Hauptprogramm (wahrscheinlich wegen des Debuggers) einen erweiterten »Rahmencode«, der zusätzlich das BP-Register auf den Stack bringt bzw. wieder herunterholt:

```
debug null.exe
-u 0,d
4859:0000 9A00005B48    CALL   485B:0000        ; SYSTEM: Initialisierung
4859:0005 55            PUSH   BP          <-- eingefügt
4859:0006 89E5          MOV    BP,SP            ; "begin"
4859:0008 89EC          MOV    SP,BP            ; "end"
4859:000A 5D            POP    BP          <-- eingefügt
4859:000B 31C0          XOR    AX,AX            ; Exitcode 0
4859:000D 9AD8005B48    CALL   485B:00D8        ; Ende über Aufruf von HALT
```

Weil das Hauptprogramm mit diesen beiden zusätzlichen Befehlen nicht mehr $10, sondern $12 Bytes umfaßt, verschiebt sich der Beginn des Codesegments von *System* um ein ganzes Segment:

```
Start  Stop   Length Name          Class

00000H 00011H 00012H NULL          CODE
00020H 004A5H 00486H SYSTEM        CODE
004B0H 00733H 00284H DATA          DATA
00740H 0473FH 04000H STACK         STACK
04740H 04740H 00000H HEAP          HEAP
```

Ein weiterer und erstaunlicher Unterschied: Obwohl *System* in der Version 5.0 um einige Grundfunktionen erweitert wurde, ist der Codeumfang um rund $70 Bytes gegenüber der Version 4.0 geschrumpft*).

Ein Einschub zum Thema »Relozieren«

Der 8086 verwendet für *near*-Sprünge und Unterprogrammaufrufe relative Adressen. Ein Befehl wie

```
xxxx:0180  E81D00      CALL $01A0
xxxx:0183    ....
```

wird folgendermaßen ausgeführt: Der Prozessor liest den CALL-Befehl, erhöht dabei das IP-Register auf die Adresse des nächsten Befehls ($0183) und bringt diesen Wert als Rücksprungadresse auf den Stack. Das Argument des CALL-Befehls wird als Integer interpretiert und zu IP addiert (hier: $0183 + $001D ergibt $01A0) - von der sich ergebenden Adresse liest der Prozessor den nächsten Befehl.

Additionen zu IP verwenden den Datentyp *Word*, eventuelle Überläufe werden einfach ignoriert. In einem Aufruf wie

```
xxxx:0160  ...
           ...

xxxx:0180  E8DDFF      CALL $0160
xxxx:0183  ...
```

ergibt die Addition des Wertes $FFDD zum momentanen Stand von IP die Startadresse der Routine ($0183 + $FFDD = $10160). Dieser Mechanismus macht den Prozessor nicht nur von der Adresse des Codesegments, sondern auch von der Lage der Routinen innerhalb des Segments unabhängig. Anders gesagt: Das zuletzt gezeigte Beispiel funktioniert, solange der »Abstand« zwischen Aufrufpunkt und aufgerufener Routine gleich bleibt - also auch dann, wenn wir den Bereich $0160..$0183 auf eine andere Startadresse kopieren.

Analoges gilt für Sprünge mit JMP innerhalb eines Segments, die ebenfalls mit einem 16-Bit-Argument arbeiten- im Gegensatz zu »kurzen« Sprüngen nach Vergleichen (JA, JE, JZ usw.), bei denen das Argument nur 8 Bit umfaßt und als *Short-*

*) Hier hat offensichtlich jemand recht gründlich geputzt.

Int interpretiert wird. (Woraus sich ergibt, daß das Ziel eines »kurzen« Sprungs maximal 128 Bytes vor bzw. 127 Bytes hinter dem momentanen Punkt liegen darf).

Für Sprünge und Aufrufe, bei denen nicht nur IP, sondern auch das CS-Register verändert wird, verzichtet der 8086 auf Additionen und gibt das Sprungziel direkt an - ein Befehl wie

```
xxxx:0180  9A7799AABB  CALL FAR BBAA:9977
```

enthält die Zieladresse in direkter Form. Da die Segmentadressen des Programms aber erst beim Laden in den Hauptspeicher festgelegt werden können, ist ein derartiger Befehl in spezieller Form in der .EXE-Datei gespeichert. Zum einen zeigt ein Index der Relozierungstabelle darauf, zum anderen gibt der Segment-Anteil der Zieladresse den »Segmentabstand« zum ersten Codesegment des Programms an. Folgerichtig ist der abschließende Aufruf von *Halt* in unserer Beispieldatei NULL als

```
xxxx:013B 9AF3010100    CALL    0001:01F3
```

gespeichert - das Codesegment von *System* befindet sich $10 Bytes »oberhalb« des Programms.

Der Linker kennzeichnet Segmentadressen nicht nur in *far*-Aufrufen und Sprüngen als »relocatable«, sondern auch bei ihrer direkten Verwendung als Daten. Deshalb ist es in einem Turbo Pascal-Programm mit einer Befehlsfolge wie

```
mov dx,DATA
mov ds,dx
```

möglich, das DS-Register auf das Datensegment zu setzen (das Symbol DATA ist ebenfalls als »xx Segmente vom ersten Codesegment entfernt« in der .EXE-Datei gespeichert).

Ein »typisches« Pascal-Programm

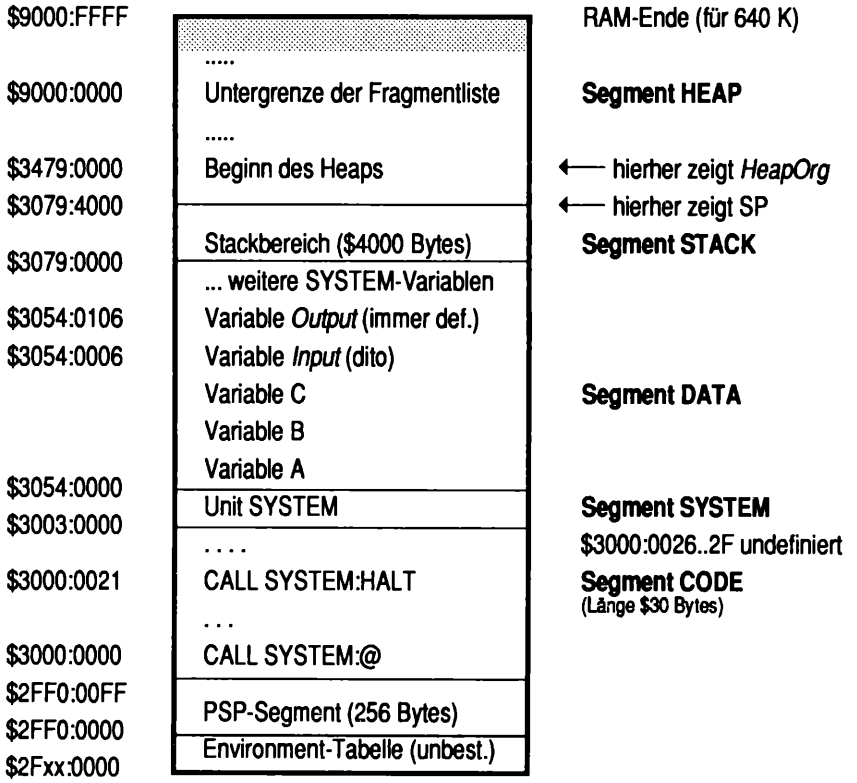
Vom Programm NUMMER1 des Benutzerhandbuchs (d.h. `Writeln('Hallo!');`) sind wir momentan noch ein gutes Stück entfernt - technisch gesehen enthält die Ausgabe einer Stringkonstanten über *Writeln* wesentlich mehr an Komplikationen, als der folgenden Analyse guttun würde. Wir beschränken uns vorläufig auf etwas noch Einfacheres - nämlich die Addition zweiter Integerzahlen:

```
program add1;
var
   A,B,C: Integer;
begin
  A := 4; B := 7;
  C := A + B;
end.
```

Da dieses Programm keine eigenen Prozeduren und Funktionen enthält, beginnt es mit der Adresse 0000 innerhalb des ersten Codesegments. Ein über SYMDEB erstelltes (und leicht nachbearbeitetes) Listing zeigt, was der Compiler aus diesen drei Zuweisungen gemacht hat:

```
0000 9A00000330     CALL   SYSTEM:@              ; Initialisierung
** begin;
0005 89E5           MOV    BP,SP                 ; V 5.0: PUSH BP / MOV BP,SP
0007 C70600000400   MOV    Word Ptr [A],0004     ; A := 4;
000D C70602000700   MOV    Word Ptr [B],0007     ; B := 7;
** C := A + B;
0013 A10000         MOV    AX,[A]                ; Wert von A -> AX
0016 03060200       ADD    AX,[B]                ; plus Wert von B
001A A30400         MOV    [C],AX                ; Speicherung in C
** end.
001D 89EC           MOV    SP,BP                 ; V 5.0: MOV SP,BP / POP BP
001F 31C0           XOR    AX,AX                 ; Exitcode 0
0021 9AF3010330     CALL   SYSTEM:HALT           ; Ende
```

Wenn wir für den ersten Befehl des Hauptprogramms die Adresse $3000:0000 annehmen, dann sieht die vollständige Struktur dieses Programms so aus:



Die Einzelheiten in der Reihenfolge aufsteigender Speicheradressen:

Hauptprogramme haben keinen eigenen Namen und werden von SYMDEB deshalb mit @ gekennzeichnet. Die Programmausführung beginnt innerhalb des Hauptprogramms von ADD1 mit einem Aufruf des Initialisierungsteils von *System* (der wiederum ein Hauptprogramm darstellt und deshalb ein @ als Namen hat). In *System* wird unter anderem das DS-Register auf das Segment DATA gesetzt.

Ein **Hinweis** für C-Programmierer: Dieser Aufbau erscheint zwar logisch, ist aber nicht zwingend - Turbo C-Programme beginnen mit dem standardisierten Startcode (C0x.OBJ), der zuerst *argc*, *argv* und *envp* setzt und schließlich *main* als Funktion *aufruft*, wobei sich *main* in den großen Codemodellen in einem anderen Segment als der Startcode befindet.

Das Hauptprogramm hat den mit **program** angegebenen Segmentnamen*) und in unserem Beispiel eine Länge von $0026 Bytes. Es wird vom Linker auf eine Segmentgrenze aufgerundet, endet also mit der Adresse $3000:002F (der Inhalt des Bereichs $3000:0026..002F ist undefiniert).

Oberhalb des Hauptprogramms (und eventueller Unterprogramme) befinden sich die Routinen der Laufzeitbibliothek (Unit *System*) in einem eigenen Segment. Unser Beispiel verwendet überhaupt keine direkten Aufrufe der Laufzeitbibliothek und enthält sozusagen eine Minimalversion von *System* mit einem Umfang von $510 Bytes.

Das Datensegment schließt sich an die Segmente aller verwendeten Units an - in unserem Beispiel folgt es also direkt auf den Code von *System*. Innerhalb des Datensegments kommen zuerst die Variablen des Programms in der Reihenfolge ihrer Deklaration, danach die Variablen benutzter Units (wie *Input* und *Output*, *HeapOrg*, *ExitProc* usw).

Auf das Datensegment folgt das Stacksegment, dessen Größe über den Compilerbefehl {$M} festgelegt ist. In unserem Beispiel hat es einen Umfang von $4000 Bytes, weil wir die Standardvorgabe {$M 16384, 0, 655360} verwendet haben.

Der Heap beginnt mit dem Ende des Stacks und wiederum mit einer eigenen Segmentadresse.

Version 5.0

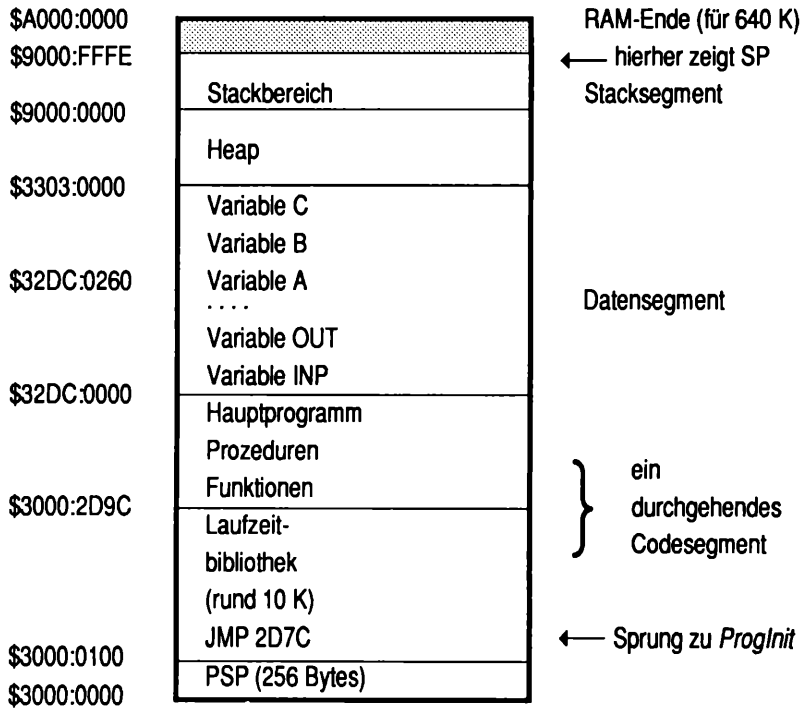
Allzugroß sind die Unterschiede hier nicht: neben dem bereits im Listing angedeuteten »Rahmencode« (PUSH BP / POP BP) kommt nur noch ein Detail hinzu, mit dem wir uns im nächsten Abschnitt noch ausführlicher beschäftigen werden: In der Version 5.0 sind die meisten globalen Variablen von *System* als typisierte Konstanten deklariert, die Variablen *A*, *B* und *C* des Hauptprogramms beginnen deshalb mit der Adresse $003C im Datensegment.

*) Wenn die entsprechende Deklaration fehlt, setzt der Linker den »Standardnamen« PROGRAM ein. .

Version 3.0

Der Compiler der Version 3.0 erzeugt .COM-Dateien, die im Gegensatz zu .EXE-Dateien von DOS als durchgehender Block behandelt werden und keinerlei Kopfinformationen enthalten. DOS setzt die Register CS, DS, ES und SS auf das Segment der »nächsten freien Adresse« und schreibt die Daten des PSP in die ersten $100 Bytes dieses Segments. SP bekommt den Wert $FFFE, danach wird das Programm mit einem Sprung zur Adresse $0100 des Segments gestartet.

Das mit der Version 3.01A erzeugte Programm ADD1.COM hat die folgende Struktur, die größtenteils erst durch die programmeigenen Initialisierungsroutinen erzeugt wird:



Abgesehen davon, daß das Programm nur ein einziges Codesegment enthält (und folglich auch keine FAR-Aufrufe), bestehen mehrere grundlegende Unterschiede in der Verteilung der Adressen:

- Das PSP befindet sich im selben Segment wie der Programmcode und belegt dort die ersten $0100 Bytes. Die Programmausführung beginnt grundsätzlich mit der Adresse $0100.
- Die Laufzeitbibliothek ist in ihrer Größe fixiert (die Adressen der Standardfunktionen etc. sind es ebenfalls) und befindet sich zwischen der Startadresse

(Codesegment:$0100) und dem durch die Compilierung erzeugten Programmcode.

- Die ersten $260 Bytes des Datensegments sind durch Variablen der Laufzeitbibliothek belegt, die erste im Programm deklarierte Variable wird ab der Adresse $0260 gespeichert.
- Der Heap schließt sich unmittelbar an das Datensegment an und *endet* mit dem Stack. Der Stack belegt die obersten 64 KByte des Systems.

Ein weiterer Unterschied: DOS nimmt keine Berechnung für die oberste verfügbare Adresse (Bytes 02/03) vor, sondern setzt die tatsächliche Endadresse des RAMs ein. Eventuelle Begrenzungen erfolgen durch die Initialisierung des Programms.

Das Programm ADD1.PAS wird vom Compiler der Version 3.01A folgendermaßen übersetzt:

```
0100 E9792C        JMP    2D7C          ; Sprung zur Initialisierung
     ....
          Laufzeitbibliothek (fix)
     ....
2D7C E857DD        CALL   0AD6   ; PROGINIT - liest die folgenden Daten:
2D7F  0006     ; "Environment-Bits" - hier: CBreak := True;
      3117     ; Codesegment des Compilers (für Rücksprung nach "Run")
      3AD7     ; Datensegment des Compilers (dito)
      02DC     ; Programmlänge in Paragraphs
      0027     ; Größe des Datensegments in Paragraphs
      0400     ; MinStack
      A000     ; MaxStack
      0010     ; Maximalzahl gleichzeitig offener Dateien
      0000     ; Recordlänge von INP
      0000     ; Recordlänge von OUT

; hierher führt der Rücksprung von PROGINIT
2D93 8BEC          MOV    BP,SP
2D95 E834DF        CALL   0CCC   ; SHIFTOVERLAY -
2D98  0025     ; ?
      2D9C     ; *** Startadresse des Programms ***
      00       ; -> keine Verschiebung

; hier beginnt der tatsächlich vom Compiler erzeugte Code
2D9C E90000        JMP    2D9F                 ; Sprung zum Hauptprogramm

2D9F B80500        MOV    AX,0004              ; A := 4;
2DA2 A36002        MOV    [A],AX
2DA5 B80700        MOV    AX,0007              ; B := 7;
2DA8 A36202        MOV    [B],AX
2DAB A16002        MOV    AX,[A]               ; C := A + B;
2DAE 03066202      ADD    AX,[B]
```

```
2DB2 A36402        MOV    [C],AX
2DB5 E90000        JMP    2DB8                    ; "end;"

2DB8 33C0          XOR    AX,AX                   ; Exitcode 0
2DBA E8CCDE        CALL   0C89                    ; Programmende
```

Units, Variablen und Konstanten

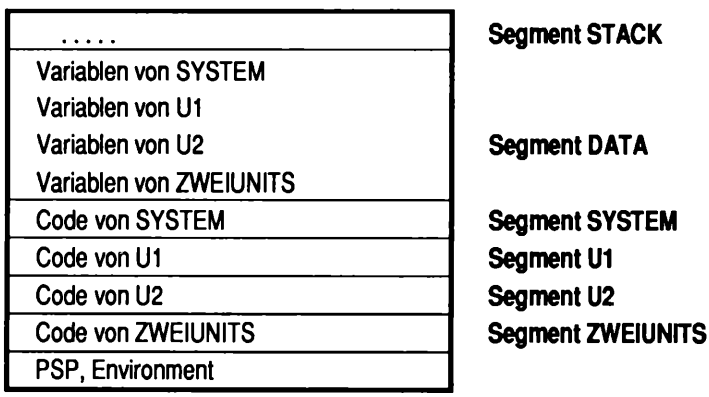
Wie in Kapitel 25 des Referenzhandbuchs beschrieben, werden Units in der umgekehrten Reihenfolge ihrer Angabe mit **uses** in ein Programm aufgenommen. Dasselbe gilt für globale Variablen von Units: sie stapeln sich in der Reihenfolge ihrer Deklaration im Datensegment. Wie eine .MAP-Datei des im letzten Abschnitt verwendeten Beispiels ADD1 zeigt, kommen die im Programm deklarierten Variablen zuerst, darauf folgen *Input*, *Output* und die restlichen Variablen von *System*:

```
 Address          Publics by Value
0000:0000        @                    ; "Name" des Hauptprogramms
0003:0000        @                    ; "Name" der Initialisierung von SYSTEM
0054:0000        A
0054:0002        B                    ; Variablen des Hauptprogramms
0054:0004        C
0054:0006        INPUT                ; Variablen von SYSTEM
    ....
```

Für ein Programm, das zwei Units explizit mit **uses** aufführt, also z.B.

```
program ZweiUnits;
uses U1, U2;
begin end.
```

ergibt sich eine Struktur, die im Prinzip so aussieht:



Die grafische Darstellung der Programmstruktur in Kapitel 25 des Referenzhandbuchs enthält zusätzlich noch den Hinweis, daß typisierte Konstanten ebenfalls im Datensegment gespeichert werden. Das folgende Beispiel zeigt, was eigentlich hinter dieser unscheinbaren Behauptung steckt:

```
program TypedConst;
uses Graph;
var
  A,B,C: Integer;
const
  ProgStr : string[6] = 'Hallo!';
begin end.
```

Das Verzeichnis der Segmente in der dazugehörigen .MAP-Datei gibt die erwartete Reihenfolge wieder: Zuerst kommt der Code des Programms, darauf folgen die Codesegmente von *Graph* und *System*, das Datensegment usw.:

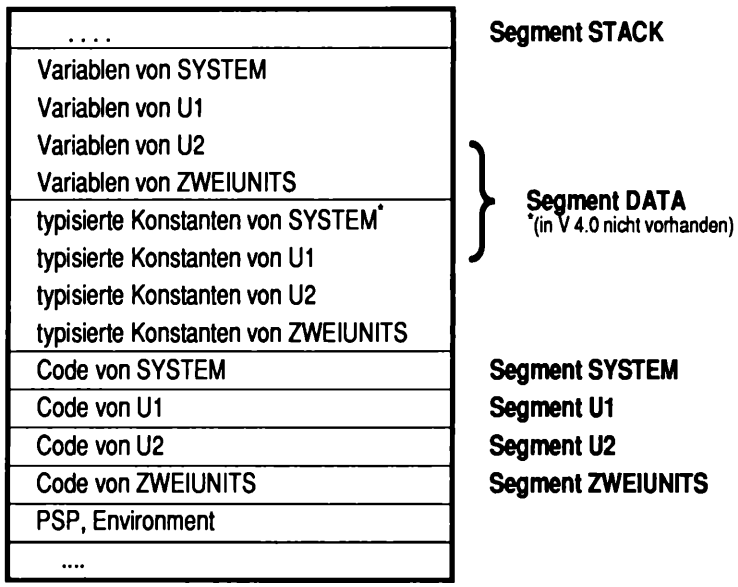
```
Start  Stop   Length Name            Class
00000H 00014H 00015H TYPEDCONST      CODE
00020H 0091BH 008FCH GRAPH           CODE
00920H 01384H 00A65H SYSTEM          CODE
01390H 019B0H 00621H DATA            DATA
019C0H 059BFH 04000H STACK           STACK
059C0H 059C0H 00000H HEAP            HEAP
```

Die Liste globaler Routinennamen enthält ebenfalls nichts Neues - aber sehen Sie sich einmal das Datensegment genauer an:

```
 Address        Publics by Value
0000:0000       @                 ; "Name" des Hauptprogramms von TYPEDCONST
--------------------------------
0002:0153       @                 ; "Name" der Initialisierung von GRAPH
0002:027E       SETVISUALPAGE
   .....                          ; die Namen der diversen von GRAPH
0002:0700       GETIMAGE           global deklarierten Routinen
--------------------------------
0092:0000       @                 ; "Name" der Initialisierung von SYSTEM
--------------------------------
0139:0000       PROGSTR           ; typisierte Konstante des Programms
                  <====           ; ein Loch mit $213 Bytes Umfang
0139:021A       A
0139:021C       B                 ; globale Variablen des Programms
0139:021E       C
0139:0220       GRAPHGETMEMPTR    ; globale Variablen von GRAPH
0139:0224       GRAPHFREEMEMPTR
                  <===            ; noch ein Loch mit $1B0 Bytes
0139:03D8       INPUT
0139:04D8       OUTPUT            ; globale Variablen von SYSTEM
    .....
```

Auf die typisierte Konstante *ProgStr*, die unabhängig von der Position ihrer Deklaration im Quelltext als erste gespeichert ist, folgt anscheinend ein Loch mit immerhin $213 Bytes Umfang - erst dann kommen die Variablen A, B und C des Hauptprogramms. Dieses Verwirrspiel - eine Art Bermuda-Dreieck für jeden, dem kein symbolischer Debugger zur Verfügung steht - hat zwei Gründe:

- Typisierte Konstanten werden auch dann vor allen anderen Variablen gespeichert, wenn sie innerhalb eines Units deklariert sind. Für ein Programm wie *ZweiUnits* ergibt sich letztendlich die folgende Struktur:



- Der Grund für das scheinbare Loch: *Graph* definiert eine Unmenge typisierter Konstanten *im Implementationsteil*. Diese Konstanten werden zwar im Datensegment gespeichert, sind aber nur innerhalb von *Graph* verfügbar - folglich erscheinen sie im (globalen) Listing der .MAP-Datei auch nicht namentlich, sondern einfach als »reservierter Bereich«.

Das zweite »Loch« in der .MAP-Datei von *TypedConst* entsteht auf ähnliche Weise: Hier handelt es sich um *Variablen*, die global innerhalb des Implementationsteils von *Graph* deklariert sind.

Version 5.0

Wie bei der Besprechung eines »typischen« Pascal-Programms bereits angedeutet, sind sämtliche globalen *System*-Variablen außer *SaveIntxx*, *Input* und *Output* hier als typisierte Konstanten deklariert und stehen deshalb am Anfang des Datensegments - die erste globale Variable des Programms erhält die Adresse $003C:

```
 Address        Publics by Value

0000:0000       @                  ; "Name" des Hauptprogramms
004C:0000       OVRCODELIST        ; Liste geladener Codesegmente
004C:0002       OVRHEAPSIZE        ; Größe des Overlay-Puffers
004C:0004       OVRDEBUGPTR        ; für den integrierten Debugger
004C:0008       OVRHEAPORG         ; Start des Overlay-Puffers
  ...
004C:0014       HEAPORG            ; Beginn des regulären Heaps
004C:0018       HEAPPTR            ; momentane Heap-Spitze
  ...
004C:0036       RANDSEED           ; Startwert für "Zufallszahlen"
004C:003A       FILEMODE           ; Datei-Eröffnungsmodus
004C:003B       TEST8087           ; "Coprozessor vorhanden"

004C:003C       A
004C:003E       B                  ; Variablen des Programms
004C:0040       C
004C:0042       INPUT              ; Variablen von SYSTEM
004C:0142       OUTPUT
004C:0242       SAVEINT00
004C:0246       SAVEINT02
 ...
```

Die Initialisierungsteile der Standard-Units

Der erste Befehl eines Pascal-Hauptprogramms besteht grundsätzlich aus einem Aufruf des Initialisierungsteils von *System*. Auf ihn folgen die Aufrufe der Initialisierungsteile weiterer aufgenommener Units - unabhängig davon, welche Routinen dieser Units später tatsächlich verwendet werden. Ein Programm, das eine uses-Anweisung wie

```
uses Crt, Dos, Printer, Graph;
```

enthält, beginnt also mit den folgenden Aufrufen (die interessanterweise diesmal in der »richtigen« Reihenfolge stehen, d.h. mit *System* an erster Stelle):

```
0000 9A0000873B     CALL   SYSTEM:@
0005 9A0000213B     CALL   DOS:@
000A 9A05001E3B     CALL   PRINTER:@
000F 9A53018E3A     CALL   GRAPH:@
                                <--   V 5.0: PUSH BP
0014 89E5           MOV    BP,SP        ; "begin" des Hauptprogramms
       ....
```

Wo ist der Aufruf des Initialisierungsteils von *Crt*? Solange auch das Unit *Dos* verwendet wird, werden Sie ihn vergeblich suchen. Offensichtlich haben die Pro-

grammierer von Borland hier ein Hintertürchen in Compiler und Linker eingebaut: *Dos* setzt *Crt* voraus und hat als einziges dieser vier Units keinen eigenen Initialisierungsteil - statt dessen wird hier die Initialisierung von *Crt* durchgeführt. Anders gesagt: Ein eigenes Segment für *Crt* und einen expliziten Aufruf dieses Initialisierungsteils bekommen Sie nur zu sehen, wenn *Crt* ohne *Dos* in ein Programm aufgenommen wird.

Diesem Hintertürchen dürfte auch ein weiterer Effekt zu verdanken sein: Wenn ein (in Pascal geschriebenes) Unit keinen eigenen Code, sondern nur Daten deklariert und an erster Stelle einer **uses**-Anweisung steht, dann übernimmt der Compiler den Namen dieses Units für *System*. Speziell in großen Programmen, die mit einem eigenen Unit für globale Datendeklarationen arbeiten, also etwa:

```
program Editor;
uses EdGlobal,     { gemeinsame und globale Variablen }
     EdEdit,       { Eingabe }
     EdPrint,      { Druckmodul }
     ....
```

wird man anfänglich verblüfft feststellen, daß der Name *System* nirgendwo erscheint - statt dessen wird am Anfang des Hauptprogramms eine Routine namens EDGLOBAL:@ aufgerufen (die mit dem Initialisierungsteil von *System* identisch ist).

Vor der Analyse der einzelnen Initialisierungsteile noch ein **Hinweis:** Obwohl diese Programmteile seit der Version 0.79 des Compilers*) (d.h. schon seit geraumer Zeit) konstant sind, ist es eher unwahrscheinlich, daß Ihre Version der Laufzeitbibliothek mit den Listings dieses Buches völlig übereinstimmt. Außerdem sind die Adressen von Variablen und verschiedenen Unterprogrammen von den Deklarationen des eigentlichen Pascal-Programms abhängig - es geht hier also nicht um Hinweise zum »Patchen«, sondern um eine allgemeine Beschreibung der ausgeführten Schritte.

Der Initialisierungsteil des Units *System*

Dieser Programmteil wird unabhängig von der Reihenfolge eventueller **uses**-Anweisungen mit dem ersten Befehl des Hauptprogramms aufgerufen und beginnt mit der Adresse 0000 innerhalb des Codesegments von *System*. Hauptsächlich werden hier vier Schritte ausgeführt:

- Setzen des DS-Registers auf das Datensegment des Programms;
- Festlegung der Eckwerte für den Heap (*HeapOrg*, *FreePtr*, *FreeMin*, *HeapError*);
- Speicherung und Veränderung diverser Interruptvektoren;

*) Gemeint ist hier die Version 4.0. So wie es aussieht, sind die Initialisierungsteile von *Printer*, *Dos* bzw. *Crt*, *Turbo3* und *Graph3* in der Version 5.0 lediglich recompiliert worden und ansonsten unverändert. Dennoch - wer die amerikanische »Politik« mit den Versionsnummern kennt...

- Initialisierung und Eröffnung der Dateivariablen *Input* und *Output*.

Dazu kommen noch einige Kleinigkeiten wie das Setzen von *PrefixSeg*, *RandSeed* und *FileMode* sowie die Festlegung einer Exit-Prozedur innerhalb von *System*, die die veränderten Interruptvektoren am Ende des Programms wieder zurücksetzt.

Ein **Hinweis zur Version 5.0:** Die zugrundeliegende Logik ist in der Version 5.0 dieselbe - allerdings kommen hier (wegen *Overlay*, dem 8087-Emulator und dem Debugger) so viele Details hinzu, daß der entsprechende Code in einem eigenen Abschnitt noch einmal gelistet ist.

```
SYSTEM:@:
0000 BAAC3B        MOV   DX,DATA            ; Konstante DATA von DOS
0003 8EDA          MOV   DS,DX              ; gesetzt: DS aufs DSeg
0005 8C06F406      MOV   [PREFIXSEG],ES     ; ES von DOS gesetzt
```

Das Register ES bekommt von DOS beim Laden des Programms die Segmentadresse des Programmsegment-Präfix zugewiesen. Neben dem Setzen des DS-Registers (auf das sich die Initialisierungsteile der restlichen Units verlassen) ist das der Hauptgrund, warum der Initialisierungsteil von *System* als allererstes beim Start eines Programms durchlaufen werden muß.

```
0009 8BC4          MOV   AX,SP
000B 051300        ADD   AX,0013
000E B104          MOV   CL,04
0010 D3E8          SHR   AX,CL              ; SP in "Paragraphs"
0012 8CD2          MOV   DX,SS              ; + Stacksegment
0014 03C2          ADD   AX,DX              ; ergibt Stack-Ende
0016 A3F806        MOV   [HEAPORG+02],AX    ; -> = Heap-Beginn
0019 A3FC06        MOV   [HEAPPTR+02],AX
```

Das Registerpaar SS:SP ist ebenfalls beim Laden des Programms von DOS gesetzt worden. Da der Initialisierungsteil von *System* mit CALL FAR aufgerufen wurde, zeigt SP momentan auf die Endadresse des Stackbereichs (= oberste Stackadresse + 1) - 4. Die Addition von $13 vor der Umrechnung in »Paragraphs« stellt sicher, daß der Heap auf jeden Fall oberhalb des Stack-Endes beginnt. Mit dem Ergebnis der Berechnung »Stacksegment (SS) + SP in Paragraphs« werden die Segment-Anteile der Variablen *HeapOrg* (Beginn des Heaps) und *HeapPtr* (momentane Spitze des Heaps) gesetzt.

```
001C 26A10200      MOV   AX,ES:[0002]       ; "Speicher-Endadresse"
0020 2D0010        SUB   AX,1000            ; minus 64 KByte
0023 A30007        MOV   [FREEPTR+02],AX    ; -> Start Fragmentliste
```

FreePtr wird mit dem von DOS im PSP gesetzten Wert für die oberste verfügbare Speicherstelle »minus 64 KByte« gesetzt. Der Offset-Anteil dieser Variablen bekommt den Wert 0000 zugewiesen und wird *vor* jedem Eintrag in die Fragmentliste um 8 erniedrigt, ohne dabei den Segment-Anteil weiter zu berücksichtigen.

Bei 640 KByte RAM (Obergrenze $A000:0000) ergibt sich für den ersten Eintrag in die Fragmentliste damit die Adresse $9000:FFF8, er belegt also die obersten 8 Byte des Speichers*).

```
0026 C7060407DD00   MOV   Word Ptr [HEAPERROR],00DD
002C 8C0E0607       MOV   [HEAPERROR+02],CS      ; HeapError^ -> SYSTEM:00DD
0030 C7060807E200   MOV   Word Ptr [EXITPROC],00E2
0036 8C0E0A07       MOV   [EXITPROC+02],CS       ; ExitProc^ -> SYSTEM:00E2
```

Als Standardvorgabe zeigt *HeapError* auf die Adresse $00DD innerhalb des Codesegments von *System*. An dieser Stelle finden sich die Befehle XOR AX,AX / RETF 0002 - diese »Fehlerbehandlung« führt also zum Abbruch des Programms mit einem Laufzeitfehler (vgl. Kapitel 25 im Referenzhandbuch). *HeapError* kann durch direkte Zuweisungen auf eine eigene Routine gesetzt werden.

ExitProc wird auf die Adresse $00E2 innerhalb des Codesegments von *System* gesetzt - hier findet sich eine Routine, die die Originalwerte veränderter Interruptvektoren wiederherstellt. Diese Zuweisung bildet den Anfang der Kette: Vor allen folgenden Veränderungen muß der Wert von *ExitProc* zwischengespeichert und im Verlauf der Exit-Prozedur wieder gesetzt werden.

```
003A C6062A0702     MOV   Byte Ptr [FILEMODE],02
003F 33C0           XOR   AX,AX                  ; Offset-Anteile der
0041 A3F606         MOV   [HEAPORG],AX             diversen Zeiger auf 0000
0044 A3FA06         MOV   [HEAPPTR],AX
0047 A3FE06         MOV   [FREEPTR],AX
004A A30207         MOV   [FREEMIN],AX
004D A31207         MOV   [RANDSEED],AX          ; RandSeed auf 00000000
0050 A31407         MOV   [RANDSEED+02],AX
0053 A32B07         MOV   [FILEMODE+01],AX       ; IOResult auf 0
```

Als Standardvorgabe für den Modus bei der Eröffnung von Dateien (*FileMode*) wird der Wert 02 gesetzt, die Offset-Anteile der diversen Heap-Zeiger bekommen den Wert 0. *HeapOrg* und *HeapPtr* zeigen danach tatsächlich auf einen Segmentanfang, für *FreePtr* ergibt sich bei der nächsten Erniedrigung ein Wert im Bereich von $FFFx (siehe oben). Die Standardvorgabe für *FreeMin* (freier Bereich zwischen der Fragmentliste und der Heap-Spitze) ist 0. Der Startwert für den Zufallsgenerator (*RandSeed*) wird hier ebenfalls explizit auf 0 gesetzt: Aus diesem Grund liefern Aufrufe von *Random* ohne ein vorheriges *Randomize* immer dieselben Zahlenfolgen. In der »unbenannten« Systemvariablen des Typs *Word* direkt hinter *FileMode* wird der mit *IOResult* ermittelbare Statuscode gespeichert - diese Variable fragt das Programm nach jeder Ein-/Ausgabeoperation ab.

*) Ein Flüchtigkeitsfehler im 4.0-Handbuch (Band II, Seite 169): Um die tatsächliche Obergrenze des Speichers zu erhalten, muß der Wert $1000 zum *Segment*-Anteil von *FreePtr* hinzuaddiert werden - und nicht zum Offset.

```
0056 B80035      MOV   AX,3500              ; Interrupt $00
0059 CD21        INT   21
005B 891E1607    MOV   [SAVEINT00],BX         -> SaveInt00
005F 8C061807    MOV   [SAVEINT00+02],ES
0063 B80235      MOV   AX,3502              ; Interrupt $02
0066 CD21        INT   21
0068 891E1A07    MOV   [SAVEINT02],BX         -> SaveInt02
006C 8C061C07    MOV   [SAVEINT02+02],ES
0070 B82335      MOV   AX,3523              ; Interrupt $23
0073 CD21        INT   21
0075 891E1E07    MOV   [SAVEINT23],BX         -> SaveInt23
0079 8C062007    MOV   [SAVEINT23+02],ES
007D B82435      MOV   AX,3524              ; Interrupt $24
0080 CD21        INT   21
0082 891E2207    MOV   [SAVEINT24],BX         -> SaveInt24
0086 8C062407    MOV   [SAVEINT24+02],ES
008A B87535      MOV   AX,3575              ; Vektor $75
008D CD21        INT   21
008F 891E2607    MOV   [SAVEINT75],BX         -> SaveInt75
0093 8C062807    MOV   [SAVEINT75+02],ES
```

Die Originalwerte der Interruptvektoren $00 (Division durch 0), $02 (NMI), $23 (CONTROL-C, CONTROL-BREAK), $24 (»Critical Error«) und $75 (Coprozessor) werden gespeichert. Der Vektor $75 wird nur dann neu gesetzt, wenn das Programm mit {$N+} compiliert wurde (siehe Kapitel 2).

```
0097 1E          PUSH  DS
0098 0E          PUSH  CS
0099 1F          POP   DS                   ; DS aufs Codesegment
009A BAE901      MOV   DX,01E9
009D B80025      MOV   AX,2500              ; INT $00 -> CS:01E9
00A0 CD21        INT   21
00A2 BAF001      MOV   DX,01F0
00A5 B82325      MOV   AX,2523              ; INT $23 -> CS:01F0
00A8 CD21        INT   21
00AA BABF01      MOV   DX,01BF
00AD B82425      MOV   AX,2524              ; INT $24 -> CS:01BF
00B0 CD21        INT   21
00B2 1F          POP   DS
```

Neusetzen der Interruptvektoren $00, $23 und $24 auf entsprechende Behandlungsroutinen innerhalb von *System*:

INT $00 wird vom Prozessor bei Divisionen durch 0 (DIV-Befehl) ausgelöst, der dabei (wie bei einem »richtigen« Interrupt) das Registerpaar CS:IP auf den Stack bringt. In diesem Spezialfall handelt es sich allerdings um die tatsächliche Adresse des DIV-Befehls (und nicht um einen Wert für einen Rücksprung). Die Routine holt diese beiden Werte wieder vom Stack, lädt das Register AX mit der Fehlernummer

$00C8 (= 200) und springt zu der Routine in *System* weiter, die Laufzeitfehlermeldungen ausgibt und das Programm danach via *ExitProc* beendet.

INT $23 wird von DOS nach der Eingabe von CONTROL-C oder CONTROL-BREAK aufgerufen und beendet das Programm nach einem Aufruf über »Control Break Exit« (PSP, Bytes $0016/17) über die *ExitProc*-Kette mit dem Exitcode 255*). Dieser Vektor bleibt auch dann auf *System* gesetzt, wenn *Crt* mit in das Programm aufgenommen wurde: *Crt* verändert den Vektor $1B und greift so einige Stufen vorher in einen *Break*-Ablauf ein.

INT $24 wird von DOS bei »kritischen Fehlern« (Klappe des Diskettenlaufwerks offen, Schreib-/Lesefehler usw.) zusammen mit einem ganzen Block von Parametern auf dem Stack aufgerufen. Die Routine in *System* prüft den Stand des DOS-Flags *Verify* über einen entsprechenden DOS-Aufruf (Funktion $54), wirft den Parameterblock vom Stack und kehrt zu DOS zurück, wo die Operation abgebrochen wird. Die eigentliche Auswertung des Fehlers findet erst über *IOResult* statt, wenn DOS zum Pascal-Programm zurückkehrt.

Diese Automatik über den Vektor $24 bringt 3.0-Programmierer und -Autoren um eines ihrer komplexesten Themen.

```
00B3 B80600     MOV   AX,0006        ; Adresse von INPUT
00B6 1E         PUSH  DS               zweimal auf den Stack:
00B7 50         PUSH  AX               1) für ASSIGN
00B8 1E         PUSH  DS               2) für RESET
00B9 50         PUSH  AX
00BA B8DC00     MOV   AX,00DC        ; CS:00DC <- Name für
00BD 0E         PUSH  CS               Input (= '')
00BE 50         PUSH  AX
00BF 0E         PUSH  CS             ; ASSIGN arbeitet mit
00C0 E81809     CALL  ASSIGN           RETF - Initialisierung
00C3 0E         PUSH  CS             ; RESET arbeitet mit RETF
00C4 E88409     CALL  RESET          ; Eröffnung
```

Initialisierung und Öffnung der Datei *Input*: Die Adresse der Dateivariablen (hier: $0006) wird zweimal auf den Stack gebracht, darüber kommt die Adresse des Dateinamens, der im *Codesegment* von *System* gespeichert ist. Auf SYSTEM:00DC findet sich ein einzelnes Byte mit dem Wert 0 - *Input* bekommt also einen Nullstring als »Namen« zugeordnet. Der Aufruf von *Assign* setzt die einzelnen Felder der übergebenen Variablen *Input* mit den folgenden Werten:

*) Diesen Exitcode hätte ich eigentlich im 5.0-Handbuch nachtragen können (Tschuldigung).

```
Handle := 0;            { wird später via RESET gesetzt }
Mode   := FmClosed;     { Datei ist nicht offen }
Private :=  0;
BufPos := 0;            { Pufferanfang }
BufEnd := 0;            { Puffer ist leer }
BufPtr := Addr(Buffer); { := Adresse von Input + 128 }
OpenFunc := <OPEN-Routine innerhalb von SYSTEM>
InOutFunc,
FlushFunc,
CloseFunc           { undefiniert }
UserFill            { undefiniert }
Name[0] := #0;      { Name: '' }
Buffer              { undefiniert }
```

Auf diese Initialisierung folgt der Aufruf von *Reset*, der die Variable *Input* für nachfolgende Leseaktionen öffnet. Sowohl *Assign* als auch *Reset* erwarten eigentlich *far*-Aufrufe und enden mit einem entsprechenden Rücksprung - deshalb wird hier vor dem Aufruf auch noch das Codesegment von *System* auf den Stack gebracht.

```
00C7 B80601     MOV   AX,0106        ; Adresse von OUTPUT
00CA 1E         PUSH  DS               zweimal auf den Stack
00CB 50         PUSH  AX               1) für ASSIGN
00CC 1E         PUSH  DS               2) für REWRITE
00CD 50         PUSH  AX
00CE B8DC00     MOV   AX,00DC        ; CS:00DC <- Name für
00D1 0E         PUSH  CS               Output (= '')
00D2 50         PUSH  AX
00D3 0E         PUSH  CS             ; ASSIGN arbeitet mit
00D4 E80409     CALL  ASSIGN           RETF - Initialisierung
00D7 0E         PUSH  CS             ; REWRITE arbeitet mit RETF
00D8 E87509     CALL  REWRITE        ; REWRITE
00DB CB         RETF
```

Derselbe Ablauf für die Dateivariable *Output*, die ebenfalls einen Nullstring als »Namen« zugewiesen bekommt und danach über *Rewrite* für Schreibaktionen eröffnet wird.

Der Initialisierungsteil von *System* endet mit der Eröffnung dieser beiden Dateien, das abschließende RETF bringt den Prozessor zum Hauptprogramm des Pascal-Programms*) zurück.

Ein **Hinweis:** Im Gegensatz zu allen anderen Units gibt *System* anscheinend nur globalen Variablen einen Namen - das gezeigte Listing ist also eine Fälschung, was die Aufrufe von *Reset*, *Rewrite* und *Assign* betrifft.

*) (Was für eine Formulierung!)

Version 5.0

Neben einer ganzen Reihe von Kleinigkeiten unterscheidet sich die Version 5.0 in einem zentralen Punkt:

In der Version 4.0 speichert jedes Unit nur die Interruptvektoren, die sein Initialisierungsteil auch tatsächlich verändert (was unter anderem zur Folge hat, daß *Crt* eine eigene Exit-Prozedur definieren muß). In der Version 5.0 speichert *System* sämtliche Interruptvektoren, die das Laufzeitsystem von Turbo Pascal benutzen *könnte* - unabhängig davon, ob *Crt*, *Overlay* und der 8087-Emulator Bestandteil des Programms sind oder nicht.

Das Zurücksetzen der Interruptvektoren am Ende des Programms wird via *Halt* erledigt*) - eine eigene Exit-Prozedur für *System* ist damit ebenfalls unnötig.

Was die Kleinigkeiten betrifft:

- neben den Zeigern für den Heap werden zusätzlich die Zeiger für den Overlay-Puffer gesetzt;
- die Statusvariable für *IOResult* hat einen »öffentlichen« Namen (*InOutRes*);
- *HeapError* wird durch *System* nicht gesetzt und hat standardmäßig den Wert **nil** (woraus auch folgt, daß die 5.0-Version von *GetMem* im Fehlerfall nicht direkt einen Aufruf via *HeapError* ausführt, sondern zuerst den Wert dieses Zeigers prüft);
- *System* setzt zusätzlich den Vektor $3F, der im Zusammenhang mit Overlays benutzt wird.

Im Zuge dieser Änderungen und Erweiterungen haben sich nicht nur die Adressen der einzelnen Befehle kräftig verschoben - die Reihenfolge der einzelnen Schritte ist ebenfalls nicht mehr dieselbe.

Das vollständige Listing:

```
SYSTEM:@  (Version 5.0)
0000 BAA448        MOV   DX,DATA            ; Konstante DATA von DOS
0003 8EDA          MOV   DS,DX              ; gesetzt: DS aufs DSeg
0005 8C063000      MOV   [PREFIXSEG],ES     ; ES von DOS gesetzt
0009 33ED          XOR   BP,BP              ; ???
000B 8BC4          MOV   AX,SP
000D 051300        ADD   AX,0013            ; + $13 Bytes
0010 B104          MOV   CL,04
0012 D3E8          SHR   AX,CL              ; SP in "Paragraphs"
0014 8CD2          MOV   DX,SS              ; + Stacksegment
0016 03C2          ADD   AX,DX              ; ergibt Stack-Ende
```

*) Da die Prozedur *SwapVectors* im Unit *Dos* deklariert ist, wird hier doppelt gemoppelt - *Halt* erledigt das Zurücksetzen der Vektoren »zu Fuß«, d.h. mit einer eigenen Schleife.

Bis auf das XOR BP,BP - dessen Sinn sich nicht so ganz ergründen ließ - ist dieser Teil gegenüber der Version 4.0 unverändert.

```
0018 A30800        MOV    [OVRHEAPORG],AX       ; Stack-Ende = Start
001B A30A00        MOV    [OVRHEAPPTR],AX         des Overlay-Puffers
001E 03060200      ADD    AX,[OVRHEAPSIZE]      ; + Overlay-Puffergröße
0022 A30C00        MOV    [OVRHEAPEND],AX       ; = Puffer-Ende
0025 A31600        MOV    [HEAPORG+02],AX         und gleichzeitig Beginn
0028 A31A00        MOV    [HEAPPTR+02],AX         des regulären Heaps
```

Hier haben wir die erste echte Erweiterung: zwischen dem Stack-Ende und dem regulären Heap liegt der Overlay-Puffer, dessen Größe über den Wert der Konstanten *OvrHeapSize* durch den Linker festgesetzt wird. (Wie im Handbuch angegeben, bekommt *OvrHeapSize* in »overlaylosen» Programmen den Wert 0 - hier fallen das Ende des Stacks und der Anfang des Heaps wie in der Version 4.0 zusammen).

```
002B 26A10200      MOV    AX,ES:[0002]          ; "Speicher-Endadresse"
002F 2D0010        SUB    AX,1000               ; minus 64 KByte
0032 A31E00        MOV    [FREEPTR+02],AX       ; -> Start Fragmentliste
```

FreePtr wird hier wie in der Version 4.0 gesetzt.

```
0035 BF3C02        MOV    DI,023C               ; Zieladresse
0038 BEDD01        MOV    SI,01DD               ; Tabelle mit Vektornummern
003B B91200        MOV    CX,0012               ; $12 (=18) Vektoren
003E 90            NOP                          ; <- mit MASM übersetzt...
003F FC            CLD
0040 2EAC          LODSB  CS:                   ; eine Vektornummer -> AL
0042 B435          MOV    AH,35                 ; "Get Interrupt Vector"
0044 CD21          INT    21
0046 891D          MOV    [DI],BX               ; DOS liefert den Vektor
0048 8C4502        MOV    [DI+02],ES              in ES:BX
004B 83C704        ADD    DI,+04                ; Zieladresse + 4
004E E2F0          LOOP   0040
```

Für die Speicherung der Originalwerte sämtlicher Interruptvektoren, die das Laufzeitsystem von Turbo Pascal verändern könnte, verwendet *System* (analog zu *Halt*) eine eigene Schleife. Ein Blick auf die entsprechende Tabelle (im Codesegment von *System*) gibt Aufschluß darüber, um welche Vektoren es sich dabei handelt:

```
-Tabelle mit den Vektornummern im Codesegment von SYSTEM
   01DD  00    - Division durch 0
         02    - NMI
         1B    - Ctrl-Break (für CRT)
         23    - Ctrl-C, Ctrl-Break
         24    - "Critical Error"
         34 35 36 37 38 39 3A 3B 3C 3D 3E  - Coprozessor
         3F    - für OVERLAY
         75    - Coprozessor-Fehler
```

Verändert werden dagegen nur vier Vektoren, wie die nächsten Befehle zeigen:

```
0050 1E        PUSH  DS
0051 0E        PUSH  CS
0052 1F        POP   DS          ; DS aufs Codesegment
0053 BACE00    MOV   DX,00CE
0056 B80025    MOV   AX,2500     ; INT $00 -> CS:00CE
0059 CD21      INT   21            (Division durch 0)
005B BAD500    MOV   DX,00D5
005E B82325    MOV   AX,2523     ; INT $23 -> CS:00D5
0061 CD21      INT   21            (^C, ^Break)
0063 BA9D00    MOV   DX,009D
0066 B82425    MOV   AX,2524     ; INT $24 -> CS:009D
0069 CD21      INT   21            ("Critical Error")
006B BAC600    MOV   DX,00C6
006E B83F25    MOV   AX,253F     ; INT $3F -> CS:00C6
0071 CD21      INT   21            (Overlays)
0073 1F        POP   DS          ; DS wieder aufs DSeg
```

Neben $00, $23 und $24 (vgl. Seite 25f) wird hier der Vektor $3F für Overlays gesetzt - er zeigt standardmäßig auf eine Routine, die das Programm mit dem Exitcode 208 (»Overlay manager not installed«) abbricht.

```
0074 B83C00    MOV   AX,003C     ; Adresse von INPUT
0077 1E        PUSH  DS            zweimal auf den Stack:
0078 50        PUSH  AX            1) für ASSIGN
0079 1E        PUSH  DS            2) für RESET
007A 50        PUSH  AX
007B B80602    MOV   AX,0206     ; CS:0206 <- Name für
007E 0E        PUSH  CS            INPUT (= '')
007F 50        PUSH  AX
0080 0E        PUSH  CS          ; ASSIGN arbeitet mit
0081 E88301    CALL  ASSIGN        RETF - Initialisierung
0084 0E        PUSH  CS          ; RESET arbeitet mit REF
0085 E8EF01    CALL  RESET       ; Eröffnung

0088 B83C01    MOV   AX,013C     ; Adresse von OUTPUT
008B 1E        PUSH  DS            zweimal auf den Stack:
008C 50        PUSH  AX            1) für ASSIGN
008D 1E        PUSH  DS            2) für REWRITE
008E 50        PUSH  AX
008F B80602    MOV   AX,0206     ; CS:0206 <- Name für
0092 0E        PUSH  CS            OUTPUT (= '')
0093 50        PUSH  AX
0094 0E        PUSH  CS          ; ASSIGN arbeitet mit
0095 E86F01    CALL  ASSIGN        RETF - Initialisierung
0098 0E        PUSH  CS          ; REWRITE arbeitet mit REF
0099 E8E001    CALL  REWRITE     ; Eröffnung
009C CB        RETF
```

Zuordnung bzw. Öffnen der Standarddateien *Input* und *Output* laufen in der Version 5.0 exakt in derselben Weise wie in der Version 4.0 ab (wobei auch hier die Namen der Routinen *Assign*, *Reset* und *Rewrite* in das Listing hineingefälscht wurden).

Das Unit *Overlay* nebst seiner Initialisierung...
wird man hier vergeblich suchen - nicht nur in eigenen Programmen (weil *Overlay* schlicht keinen eigenen Initialisierungsteil definiert), sondern auch in diesem Buch: wie im Vorwort angekündigt, nimmt die Erläuterung der entsprechenden Mechanismen und Hintergründe sowie die Möglichkeiten zu ihrer Nachkonstruktion mit der Version 4.0 einen größeren Teil des zweiten Bandes ein.

Um ganz Ungeduldige aber nicht völlig im Regen stehenzulassen, folgt hier ein kurzer Abriß in Form einer Punkteliste, der als Grundlage eigener Forschungen dienen könnte.

❑ Für jedes als Overlay eingebundene Unit erzeugen Compiler und Linker ein eigenes Segment (mit dem Namen dieses Overlay-Units), das als fester Bestandteil in das Programm eingebunden wird. Für ein Overlay-Unit wie

```
unit Ovr;
{$O+,D-}
interface
   procedure Proc1;
   procedure Proc2;
   procedure Proc3;
implementation
....
```

ergibt sich ein Segment *Ovr* mit den folgenden Daten:

```
OVR:0
0000 CD3F0000        INT    3F 0000   ; ???
0004 0004 0000       ; Position des Codes in der .OVR-Datei als LongInt
0008 0042            ; Umfang des Codes
000A 000C            ; ??
000C 0004            ; ??
000E 0000
0010 0000            ; Segmentadresse des geladenen Codes
0012 0000 0000 0000 0000 0000 0000 0000

0020 CD3FFFFF00      INT    3F FFFF 00   ; "Dummy?"
0025 CD3F000000      INT    3F 0000 00   ; Proc1 - beginnt mit Offset $0000
002A CD3F160000      INT    3F 1600 00   ; Proc2 - beginnt mit Offset $0016
002F CD3F2C0000      INT    3F 2C00 00   ; Proc3 - beginnt mit Offset $002C

; Ende des Segments OVR
```

Jede Overlay-Routine ist hier in der Form `INT $3F xxxx yy` eingetragen: *xxxx* steht für den Offset innerhalb des Codesegments, *yy* dient lediglich als Platzhalter für eine spätere Umwandlung des Interrupt-Befehls in ein »CALL FAR« (siehe übernächster Absatz).

❑ *Aufrufe* von Overlay-Routinen durch das Programm sind als »CALL FAR zum jeweiligen Interrupt« codiert - der Aufruf der Prozedur *Proc1* also als `CALL FAR OVR:0025`, der Aufruf von *Proc2* als `CALL FAR OVR:002A` usw.

❑ Der Interruptvektor $3F wird durch den Initialisierungsteil von *System* auf eine Routine gesetzt, die das Programm mit dem Exitcode 208 (»Overlay manager not installed«) abbricht. Die Routine *OvrInit* setzt diesen Vektor erneut - und zwar auf einen Dispatcher (»Funktionsverteiler«), der die folgenden Schritte ausführt:

1. Prüfung, ob der Overlay-Puffer noch genug Platz für das dazugehörige Codesegment bietet. Wenn nein: Hinauswerfen anderer Overlay-Codesegmente (siehe Punkt 4).
2. Laden des Codesegments in den Overlay-Puffer.
3. Ersatz sämtlicher INT $3F-Befehle dieses Segments durch CALL FAR. Aus

   ```
   INT $3F xxxx yy
   ```

   wird

   ```
   CALL FAR <Codesegment-Ladeadresse>:xxxx
   ```

   (Wenn der Code von *Ovr* also beispielsweise ab $4740:0000 in den Overlay-Puffer geladen wurde, dann wird der Aufruf von *Proc2* (`INT $3F 0016 00`) durch `CALL FAR 4740:0016` ersetzt).
4. Ab diesem Punkt ist die Overlay-Verwaltung bei Aufrufen von Routinen des neu geladenen Units nicht mehr beteiligt - sie hat sich mit dem Ersatz der INT $3F-Befehle durch CALL FAR sozusagen selbst hinausgeworfen. Ins Spiel kommt sie erst dann wieder, wenn ein weiteres Overlay-Unit geladen werden soll, der Puffer für beide Units zusammen nicht reicht und deshalb ein zuvor geladenes Unit seinen Platz räumen muß: In diesem Fall werden die CALL FAR-Befehle wieder in `INT $3F <Offset> <yy>` verwandelt.

Einige weitere Einzelheiten zum momentanen Stand der Forschungen: *OvrCodeList* und *OvrLoadList* stellen jeweils den Beginn einer Zeigerliste dar, deren Elemente die Segmentadressen der momentan geladenen Codesegmente bzw. die Segmentadressen der (fix eingebundenen) Overlay-Units mit momentan geladenen Codesegmenten enthalten; eine »Eigenintelligenz« beim Entfernen geladener Overlay-Units (wie beispielsweise Frequenzanalyse oder LRU-Technik) war nicht auszumachen*).

*) Sowohl die amerikanische als auch die deutsche Ausgabe des Benutzerhandbuchs übertreiben hier also ein wenig - die »Eigenintelligenz« von *Overlay* liegt hauptsächlich im Grundprinzip. (Das allerdings ist so gerad-

Der Initialisierungsteil von *Crt* bzw. *Dos*

Wie bereits gesagt, setzt *Dos* das Unit *Crt* voraus und hat selbst keine eigene Initialisierung - etwas irritierend ist hier nur, daß *Crt* bei der Aufnahme von *Dos* kein eigenes Codesegment besitzt. (Wahrscheinlich sieht die Sache so aus, daß *Dos* sämtliche Deklarationen und den Initialisierungsteil von *Crt* noch einmal enthält und Compiler bzw. Linker das Unit *Crt* bei der Verwendung von *Dos* komplett ausklammern). Wie dem auch immer sei - die Initialisierung von *Crt* besteht hauptsächlich aus fünf Schritten:

- Prüfung des gesetzten Videomodus, Festlegen von *LastMode*, *CheckSnow*, *DirectVideo* und *TextAttr*;
- Aufruf von *AssignCrt* mit *Input* und ein nachfolgendes *Reset*;
- Aufruf von *AssignCrt* mit *Output* und ein nachfolgendes *Rewrite*;
- Zeitmessung über den »Clock Tick« für *Delay*;
- Setzen des Interruptvektors $1B (*Break*) auf eine eigene Routine und entsprechende Erweiterung der *ExitProc*-Kette, damit dieser Vektor beim Ende des Programms wieder seinen Originalwert zurückbekommt.

Der Initialisierungsteil von *Crt* beginnt wie der von *System* mit der Adresse 0000 innerhalb des entsprechenden Codesegments.

```
CRT:@:
0000 89E5          MOV    BP,SP              ; V 5.0: PUSH BP/MOV BP,SP
0002 E82900        CALL   002E               ; DirectVideo usw.
```

Die Initialisierung von *DirectVideo*, *CheckSnow* usw. ist als Unterprogramm gehalten, wird aber nur einmal verwendet. »Strukturierte Programmierung«?

```
0005 BF0600        MOV    DI,0006            ; Adresse von INPUT
0008 1E            PUSH   DS                   auf den Stack
0009 57            PUSH   DI
000A 0E            PUSH   CS                 ; AssignCrt arbeitet
000B E86B03        CALL   ASSIGNCRT            mit RETF
000E BF0600        MOV    DI,0006
0011 1E            PUSH   DS
0012 57            PUSH   DI
0013 9A4B0AA23A    CALL   SYSTEM:RESET       ; Input.Mode = fmInput
```

Der Aufruf von *AssignCrt* setzt *Mode* := *fmInput*, *BufSize* := (dezimal) 128, *BufPtr* auf die Adresse von *Input* + 128 und *OpenFunc* auf eine Routine innerhalb von *Crt*. Diese Routine wird über den nachfolgenden Aufruf von *Reset* aktiviert und setzt ihrerseits die *Crt*-eigene Eingabeprozedur im Feld *InOutProc*.

Ein **Hinweis:** Auch wenn im Referenzhandbuch bereits steht, daß weder *Assign* noch *AssignCrt* auf offene Dateien angewendet werden dürfen, sollte noch einmal

linig und klar strukturiert, daß man sich wieder einmal fragt, warum nicht schon längst jemand anderes draufgekommen ist...)

klargestellt werden, daß ein mit *SetTextBuf* gesetzter Puffer diese Prozeduren nicht übersteht.

Der Aufruf von *AssignCrt* ist im Gegensatz zum Aufruf von *Reset* nicht von mir ins Listing eingesetzt worden - wie bereits gesagt, haben global verfügbare Routinen nur innerhalb von *System* keinen eigenen Namen.

```
0018 BF0601        MOV   DI,0106          ; Adresse von OUTPUT
001B 1E            PUSH  DS                 auf den Stack
001C 57            PUSH  DI
001D 0E            PUSH  CS               ; AssignCrt arbeitet
001E E85803        CALL  ASSIGNCRT          mit RETF
0021 BF0601        MOV   DI,0106
0024 1E            PUSH  DS
0025 57            PUSH  DI
0026 9A500AA23A    CALL  SYSTEM:REWRITE   ; Output.Mode = fmOutput
002B 89EC          MOV   SP,BP            ; V 5.0: MOV SP,BP / POP BP
002D CB            RETF
```

Derselbe Ablauf mit *Output*: »Schließen« der Datei, Zurücksetzen des Textpuffers, Eintrag einer *Crt*-eigenen Routine in *OpenFunc* und nachfolgende Eröffnung via *Rewrite*.

AssignCrt ist im Interfaceteil von *Crt* deklariert und erwartet normalerweise einen *far*-Aufruf. Deshalb wird hier in beiden Fällen zusätzlich das CS-Register auf den Stack gebracht.

Das abschließende RETF bringt den Prozessor zum Hauptprogramm des Pascal-Programms zurück.

```
002E B40F          MOV   AH,0F            ; "Get Video Mode"
0030 E82006        CALL  0653             ; INT $10-Aufruf
0033 3C07          CMP   AL,07            ; Videomodus 7?
0035 740A          JZ    0041
0037 3C03          CMP   AL,03            ; Videomodus <= 3?
0039 7606          JBE   0041
003B B80300        MOV   AX,0003          ; weder 0..3 noch 7:
003E E87800        CALL  00B9               -> Modus 3 setzen
0041 E8D700        CALL  011B             ; LastMode, CheckSnow...
```

Wenn der momentan gesetzte Videomodus nicht im Bereich von 0..3 liegt und auch keine Monochrom- oder Hercules-Karte (Videomodus 7) verwendet wird, dann schiebt *Crt* einen Aufruf der Routine 00B9 ein, die den Videomodus 3 über einen BIOS-Aufruf setzt. (Diese Routine scheint eine Art Alleskönner zu sein - je nach Funktionsnummer setzt sie auch noch die Cursorgröße, spricht den Controller direkt an usw). Außerdem enthält sie einen INT $10-Aufruf mit dem Funktionscode $11, über den auf der übernächsten Seite geschimpft wird.

Die Routine 011B, die auf der übernächsten Seite gelistet ist, wird in jedem Fall aufgerufen; sie setzt *DirectVideo*, *LastMode*, *CheckSnow* etc.

```
0044 B408        MOV   AH,08            ; Service 8: Read Char
0046 32FF        XOR   BH,BH            ; Seite 0
0048 E80806      CALL  0653             ; INT $10
004B 8AC4        MOV   AL,AH            ; Attribut des Zeichens
004D 247F        AND   AL,7F            ; Blink-Bit raus
004F A2F104      MOV   [04F1],AL        ; ???
0052 A2E004      MOV   [TEXTATTR],AL    ; -> Standard-Attribut
```

TextAttr wird mit dem Attribut des Zeichens gesetzt, auf dem sich der Cursor momentan befindet. Dabei dürfte es sich meist um ein Leerzeichen handeln - wenn zuvor über 00B9 ein neuer Videomodus gesetzt wurde, ist das grundsätzlich der Fall. Das »Blink-Attribut« wird ausmaskiert.

```
0055 33C0        XOR   AX,AX
0057 A2DB04      MOV   [CHECKEOF],AL      ; False
005A A2F204      MOV   [04F2],AL          ; ?
005D A2F304      MOV   [04F3],AL          ; ??
0060 40          INC   AX
0061 A2DA04      MOV   [CHECKBREAK],AL    ; True
```

Die Standardvorgabe für *CheckEof* ist *False*, die Standardvorgabe für *CheckBreak* ist *True*. Was die beiden anderen Speicherzellen angeht...

```
0064 B84000      MOV   AX,0040
0067 8EC0        MOV   ES,AX
0069 BF6C00      MOV   DI,006C
006C 268A05      MOV   AL,ES:[DI]         ; Clock Tick
006F 263A05      CMP   AL,ES:[DI]         ; Warten auf Wechsel
0072 74FB        JZ    006F
0074 268A05      MOV   AL,ES:[DI]
0077 B9FFFF      MOV   CX,FFFF
007A E88C02      CALL  DELAY+20           ; Zeitmessung, zählt
007D B83700      MOV   AX,0037              CX herunter
0080 91          XCHG  AX,CX
0081 F7D0        NOT   AX                 ; Anzahl Schleifendurchläufe
0083 33D2        XOR   DX,DX                (negativ) * - 1
0085 F7F1        DIV   CX                 ; geteilt durch 55 (dezimal)
0087 A3ED04      MOV   [04ED],AX          ; DELAY-Faktor
```

Die Messung der Arbeitsgeschwindigkeit des Prozessors für *Delay*: Auf der Adresse $40:006C..6F zählt das BIOS die Anzahl der »Clock Ticks« seit dem Start des Systems. Dieser Zähler wird rund 18.2 mal in der Sekunde erhöht - wenn wir also einen Wechsel abwarten und danach mitzählen, wieviele Durchläufe einer Verzögerungsschleife der Prozessor in der Zwischenzeit ausgeführt hat, können wir darüber einen (direkt proportionalen) Zählfaktor für *Delay* festlegen (der verwendete Divisor $0037 entspricht ziemlich genau dem Wert 1 / 18.2).

(In der Version 3.0 war diese Messung noch auf eine ungeheuer komplizierte und nicht so ganz einsichtige Weise gelöst, nämlich durch den zeitweiligen Austausch des Interruptvektors $08).

```
008A B81B35        MOV   AX,351B
008D CD21          INT   21                  ; GetIntVec($1B)
008F 891EE504      MOV   [SAVEINT1B],BX
0093 8C06E704      MOV   [SAVEINT1B+02],ES
0097 1E            PUSH  DS
0098 0E            PUSH  CS
0099 1F            POP   DS
009A BA7A01        MOV   DX,017A             ; INT $1B auf CRT:017A
009D B81B25        MOV   AX,251B
00A0 CD21          INT   21
00A2 1F            POP   DS
00A3 C4060807      LES   AX,[EXITPROC]
00A7 A3E904        MOV   [SAVEINT1B+04],AX   ; alten Wert von
00AA 8C06EB04      MOV   [SAVEINT1B+06],ES     EXITPROC speichern
00AE C70608076301  MOV   [EXITPROC],0163
00B4 8C0E0A07      MOV   [EXITPROC+02],CS    ; EXITPROC auf CRT:0163
00B8 C3            RET
```

Der Austausch des Interruptvektors $1B: Dieser Vektor wird (über den Interrupt-Controller) von der Tastatur bei der Eingabe von CONTROL-BREAK benutzt. *Crt* setzt ihn auf eine eigene Routine, die eine Reaktion auf CONTROL-BREAK vom Stand des Flags *CheckBreak* abhängig macht.

Wie bereits beim Listing von *System* angemerkt, bekommt DOS (und damit *System* über den Interruptvektor $23) nach dieser Aktion von CONTROL-BREAK nur dann noch etwas mit, wenn *CheckBreak* auf *True* gesetzt ist und die entsprechende Routine innerhalb von *Crt* den Unterbrechungsversuch explizit an DOS weitergibt - ansonsten wird diese Eingabe sozusagen bereits an der Quelle ignoriert.

Damit dieser Vektor $1B am Ende des Programms wieder auf seinen Originalwert gesetzt werden kann, fügt *Crt* der *ExitProc*-Kette eine eigene Routine hinzu, die hier der Vollständigkeit halber ebenfalls gelistet ist:

```
*** ExitProc von Crt
0163 1E            PUSH  DS                  ; INT $1B zurück
0164 C516E504      LDS   DX,[SAVEINT1B]      ; -> DS:DX
0168 B81B25        MOV   AX,251B             ; "Set Interrupt"
016B CD21          INT   21                  ; DOS-Aufruf
016D 1F            POP   DS
016E C406E904      LES   AX,[SAVEINT1B+04]   ; alter Wert von EXITPROC
0172 A30807        MOV   [EXITPROC],AX       ; wird wieder eingesetzt
0175 8C060A07      MOV   [EXITPROC+02],ES
0179 CB            RETF                      ; zurück zu SYSTEM
```

So ungefähr sieht jede Exit-Prozedur aus: Zuerst kommt die auszuführende Aktion (Zurücksetzen des Interruptvektors $1B auf den Originalwert), danach wird *ExitProc* der Wert zugewiesen, den diese Variable vor dem »Einbau« der Exit-Prozedur hatte.

```
011B B40F        MOV   AH,0F          ; "Get Video Mode"
011D E83305      CALL  0653           ; INT $10
0120 50          PUSH  AX             ; AL=Modus, AH=Zeilenbreite
0121 B83011      MOV   AX,1130        ; Subfunktion $11 ??
0124 B700        MOV   BH,00            liefert offensichtlich
0126 B200        MOV   DL,00            DL = Zeilenzahl -1
0128 E82805      CALL  0653           ; INT $10
012B 58          POP   AX             ; Modus, Zeilenbreite
012C B100        MOV   CL,00
012E 0AD2        OR    DL,DL          ; für DL <> 0:
0130 7508        JNZ   013A             -> CheckSnow := False
0132 B218        MOV   DL,18          ; Zeilenzahl direkt setzen
0134 3C03        CMP   AL,03          ; Modus > 3?
0136 7702        JA    013A             -> ja, False
0138 B101        MOV   CL,01          ; ansonsten CheckSnow := True
```

Der Aufruf über INT $10 mit der Funktionsnummer $0F liefert die Zeilenbreite (80) in AH und den momentan gesetzten Videomodus in AL zurück.

Der Aufruf über INT $10 mit der Funktionsnummer $11 ist anscheinend nur beim EGA- bzw. VGA-BIOS definiert: Er liefert die Zeilenzahl minus 1 (= $0018 oder $002A für EGA, $0018 oder $0031 für VGA) im Register DL und den Wert $000E im Register CX - zumindest tut er das bei einer GENOA EGA und bei Paradise-Karten. Obwohl diese Funktion offensichtlich von mehreren Karten unterstützt wird, schweigt sich die dazugehörige Dokumentation aus - und den Autoren der diversen Zeitschriftenartikel über EGA-Karten ist auch nicht mehr als die Vogel-Strauß-Technik dazu eingefallen*). Wie dem auch sei: Beide Karten setzen das Register DX im CGA-Emulationsmodus auf den Wert 0.

Wenn auf diese Weise eine CGA-Karte erkannt wird, setzt *Crt* die Zeilenzahl direkt auf 24 und prüft, ob ein Textmodus (Mode = 0..3) aktiv ist: Wenn ja, bekommt *CheckSnow* den Wert *True*.

```
013A 8AF2        MOV   DH,DL          ; $18 oder höher
013C 8AD4        MOV   DL,AH          ; Zeilenzahl dazu
013E FECA        DEC   DL             ; auf $17 oder höher
0140 B400        MOV   AH,00          ; LASTMODE := 0 für
0142 80FE18      CMP   DH,18          ; <= 24 Zeilen,
0145 7602        JBE   0149
0147 B401        MOV   AH,01          ; sonst LASTMODE := 1
0149 A3DE04      MOV   [LASTMODE],AX  ; "letzter Videomodus"
014C 8916EF04    MOV   [04EF],DX      ; Breite/Zeilenzahl-2
0150 880EDD04    MOV   [CHECKSNOW],CL
0154 C606DC0401  MOV   Byte Ptr [DIRECTVIDEO],01 ; immer True
```

*) Tatsache ist: Ich habe von einem runden Dutzend netter Leute mittlerweile die Unterlagen »ihrer« Karte zugeschickt bekommen - und damit den Beweis, daß hier fast jeder Hersteller (GENOA, Paradise, ATI... und nicht zuletzt IBM selbst bei VGA-Karten) sein eigenes Süppchen kocht.

```
0159 33C0          XOR    AX,AX
015B A3E104        MOV    [WINDMIN],AX          ; linke obere Ecke des
015E 8916E304      MOV    [WINDMAX],DX            Textfensters: 0,0
0162 C3            RET
```

Wenn der momentan gesetzte Videomodus mehr als 24 Zeilen definiert, bekommt *LastMode* den Wert 1, ansonsten wird diese Variable auf 0 gesetzt. Was *Crt* mit der Speicherung der Zeichenzahl pro Zeile, kombiniert mit der Zeilenzahl-2 anfängt, war nicht zu ermitteln.

Zurück in erforschtere Gebiete: *DirectVideo* wird unabhängig von *CheckSnow* immer auf *True* gesetzt, die linke obere Ecke des Textfensters auf die Koordinaten 0,0. Wenn die Routine durch den Initialisierungsteil von *Crt* aufgerufen wurde, führt der Rücksprung zu CRT:0044.

Version 5.0

Abgesehen von den bereits im Listing angedeuteten PUSH- und POP-Befehlen für das BP-Register und einigen verschobenen Adressen (Setzen von Videomodus 3: $0099; Setzen von *LastMode*, *CheckSnow*: $00E7) gibt es hier nur einen Unterschied: da das Speichern und Zurücksetzen des Vektors $1B im Initialisierungsteil von *System* bzw. via *Halt* geschieht, wurden die entsprechenden Befehle, das Umsetzen von *ExitProc* sowie die Exit-Prozedur selbst aus *Crt* herausgekürzt. Sonst bleibt alles beim alten.

Der Initialisierungsteil von *Printer*

Das Unit *Printer* besteht aus drei Teilen: der Variablendeklaration *Lst*, einer Stringkonstanten mit dem Inhalt 'LPT1' und dem Initialisierungsteil, der hier gelistet ist (endlich einmal etwas Einfaches und Übersichtliches):

```
PRINTER:@:
0005 89E5          MOV    BP,SP                 ; V 5.0: PUSH BP / MOV BP,SP
0007 BFD403        MOV    DI,03D4               ; Adresse von Lst
000A 1E            PUSH   DS
000B 57            PUSH   DI
000C BF0000        MOV    DI,0000               ; <04>'LPT1'
000F 0E            PUSH   CS                      steht auf CS:0000
0010 57            PUSH   DI
0011 9ADB09A23A    CALL   SYSTEM:ASSIGN         ; initialisiert die
0016 BFD403        MOV    DI,03D4                 Variable Lst
0019 1E            PUSH   DS
001A 57            PUSH   DI
001B 9A500AA23A    CALL   SYSTEM:REWRITE        ; REWRITE
0020 89EC          MOV    SP,BP                 ; V 5.0: MOV BP,SP / POP BP
0022 CB            RETF
```

Die Initialisierung von *Lst* über *Assign* und das anschließende *Rewrite* laufen genauso ab wie die Eröffnung von *Output* im Unit *System*. Zwei Fragen wären hier allerdings zu klären (wenn Sie das Listing genau untersuchen, sind es drei):

- Wo ist die spezielle Ausgaberoutine für *Lst*, d.h. wie gelangen Ausgaben über *Lst* zur Druckerschnittstelle? DOS gibt Dateien mit dem reservierten Namen LPT1 bei der Eröffnung den Handle (= Referenznummer) für die Standard-Druckerschnittstelle und leitet Ausgaben zu dieser Datei an LPT1 weiter. Daraus ergibt sich die nächste Frage:
- Wieso behauptet das Benutzerhandbuch in Kapitel 8 bei der Besprechung von UPGRADE, daß die reservierten Dateinamen CON:, LST: usw. ab der Version 4.0 nicht mehr unterstützt werden? Was der Compiler tatsächlich nicht mehr mag, sind Doppelpunkte: Die Dateinamen CON, LPT1, LPT2 und AUX sind nach wie vor für die entsprechenden Geräte reserviert, weil sie von DOS auf spezielle Weise behandelt werden. Für OUT, TRM und LST erzeugt Turbo Pascal dagegen normale Diskettendateien.
- Wieso beginnt der Initialisierungsteil von *Printer* mit der Adresse 0005 innerhalb des Segments? Weil die Stringkonstante 'LPT1' (mit einem vorangestellten Längenbyte) im Bereich CS:0000..0004 gespeichert ist. Zu diesem Thema gibt es einen ganzen Abschnitt (»Speicherung von Variablen und Konstanten«).

Im Gegensatz zu den Initialisierungsteilen von *System* und *Crt* bzw. *Dos* ist das »Hauptprogramm« von *Printer* offensichtlich in Pascal geschrieben worden - anders wären die unnötigen Manipulationen mit BP und SP am Anfang und Ende der Routine nicht zu erklären.

Ein **Hinweis:** Weil die Ausgabe an einen Drucker über DOS und/oder BIOS auch beim AT noch unglaublich langsam arbeitet und sich die Druckerschnittstelle hervorragend für die Demonstration eigener Gerätetreiber eignet, gibt es auch dazu noch einen eigenen Abschnitt (»FastList« in Kapitel 6).

Version 5.0

Abgesehen von den bereits im Listing angedeuteten PUSH- und POP-Befehlen für das BP-Register gibt es keine Unterschiede zur Version 4.0.

Der Initialisierungsteil von *Graph*

Graph ist nicht nur ein komplettes Grafikpaket mit allen denkbaren Funktionen, sondern auch das komplexeste Unit mit allen vorstellbaren Absonderlichkeiten. Das fängt damit an, daß *Graph* rund $200 Bytes des Datensegments mit typisierten Konstanten belegt, geht mit weiteren rund $200 Variablen-Bytes weiter, von denen nur zwei einen Namen haben (*GraphGetMemPtr* und *GraphFreeMemPtr*), und hört noch lange nicht damit auf, daß der Initialisierungsteil irgendwo inmitten des Codesegments liegt (welches Codesegments? *Graph* hat insgesamt drei davon).

Da versteht es sich fast schon von selbst, daß `GRAPH:@` bei der Aufnahme weiterer Routinen kräftig verschoben wird - in der gezeigten Form bekommen Sie diese Routine nur zu sehen, wenn Sie ein Programm schreiben, das nur aus **begin** und **end** besteht, also:

```
uses Graph;
begin
end.
```

Wir gehen hier auch nur auf die allerwichtigsten Dinge ein (der Rest folgt in einem eigenen Kapitel):

- Erhöhung von *FreeMin* auf 256 Bytes;
- Installation einer eigenen Routine zur Behandlung von Heap-Fehlern (über das Setzen von *HeapError*);
- Setzen von *GraphGetMemPtr* und *GraphFreeMemPtr* - den einzigen »öffentlichen« Variablen, die *Graph* deklariert.

Außer *FreeMin* und *HeapError* werden keine Variablen anderer Units verändert, Interruptvektoren und/oder Exit-Prozeduren setzt *Graph* ebenfalls nicht. Tatsächlich besteht die gesamte Routine aus einer Unmenge von Initialisierungen »privater« Variablen, von denen rund die Hälfte als typisierte Konstanten deklariert ist. (Um es genau zu nehmen: Alle Variablen, die in diesem Listing mit einer niedrigeren Adresse als *GraphGetMemPtr* und *GraphFreeMemPtr* erscheinen).

```
GRAPH:@:
0153 89E5          MOV   BP,SP                    ; V 5.0: PUSH BP/MOV BP,SP
0155 31C0          XOR   AX,AX
0157 A34C03        MOV   [034C],AX
015A C706D8050001  MOV   Word Ptr [FREEMIN],0100 ; 256 Bytes Minimum
0160 C70603030010  MOV   Word Ptr [0303],1000
0166 0E            PUSH  CS
0167 E86902        CALL  03D3                     ; viele Initialisierungen
016A E87CFF        CALL  00E9                     ; noch mehr davon
016D B83F01        MOV   AX,013F
0170 8CCA          MOV   DX,CS                    ; HEAPERROR auf
0172 A3DA05        MOV   [HEAPERROR],AX             GRAPH:013F, entfällt
0175 8916DC05      MOV   [HEAPERROR+02],DX          für Version 5.0
0179 B8A500        MOV   AX,00A5
017C 8CCA          MOV   DX,CS
017E A31202        MOV   [GRAPHGETMEMPTR],AX      ; GRAPHGETMEMPTR auf
0181 89161402      MOV   [GRAPHGETMEMPTR+02],DX     GRAPH:00A5
0185 B8BB00        MOV   AX,00BB
0188 8CCA          MOV   DX,CS
018A A31602        MOV   [GRAPHFREEMEMPTR],AX     ; GRAPHFREEMEMPTR auf
018D 89161802      MOV   [GRAPHFREEMEMPTR+02],DX    GRAPH:00BB
```

Graph lädt normalerweise einen Grafiktreiber auf den Heap und greift deshalb in die dynamische Speicherverwaltung ein: Der minimale Platz zwischen der Heap-

Spitze und dem unteren Ende der Fragmentliste (*FreeMin*) wird von 0 (*System*-Standardvorgabe) auf 256 Bytes erhöht, durch das Setzen des Zeigers *HeapError* wird eine eigene Behandlungsroutine für Heap-Fehler installiert. *GraphGetMemPtr* und *GraphFreeMemPtr* sind erstaunlicherweise nicht als typisierte Konstanten deklariert und müssen deshalb bei der Initialisierung direkt gesetzt werden - wahrscheinlich deshalb, weil sich in diesem Unit bei der Aufnahme weiterer Routinen *alles* verschiebt.

Die Routinen 00E9 und 03D3 bestehen nur aus Initialisierungen »privater« Variablen und Konstanten.

```
0191 C6069C0200     MOV    Byte Ptr [029C],00
0196 C606A50200     MOV    Byte Ptr [02A5],00
019B BF0503         MOV    DI,0305
019E 1E             PUSH   DS                ; Zieladresse: DS:0305
019F 57             PUSH   DI
01A0 B81800         MOV    AX,0018           ; $18 Bytes
01A3 50             PUSH   AX
01A4 B000           MOV    AL,00             ; Wert 00
01A6 50             PUSH   AX
01A7 9A510A003B     CALL   SYSTEM:FILLCHAR
01AC 89EC           MOV    SP,BP             ; V 5.0: MPV SP,BP / POP BP
01AE CB             RETF
```

Zu guter Letzt wird noch ein Bereich von $18 Bytes Größe via *FillChar* mit dem Wert 00 gefüllt. Das (unnötige) Setzen von BP bzw. SP am Anfang und Ende der Routine deutet wie beim Unit *Printer* darauf hin, daß die Routine in Pascal geschrieben worden ist - verwendet wird BP jedenfalls nirgendwo.

Version 5.0

Hier gibt es eine echte Falltür zu vermelden, die die Entwickler von Borland eingebaut haben: In der Version 4.0 setzt der Initialisierungsteil von *Graph* über eine entsprechende Zuweisung an *HeapError* eine eigene Routine zur Behandlung von Fehlern bei der dynamischen Belegung von Speicherplatz. Dieselbe Routine findet auch in der Version 5.0 Verwendung - nur wird *HeapError* hier nicht »statisch« durch den Initialisierungsteil gesetzt, sondern jeweils direkt vor bzw. nach Aufrufen via *GraphGetMemPtr*.

Etwas kürzer gesagt: eigene Routinen für Heap-Fehler und entsprechende Zuweisungen an *HeapError* haben auf das Grafikpaket der Version 5.0 keine Wirkung*).

*) Was im Prinzip keine schlechte Idee ist: schließlich hat der Programmierer hier über eine eigene Version von *GraphGetMem* alle Freiheiten, die er sich nur wünschen kann. Wenn solche Geniestreiche nun auch noch Eingang in die Dokumentation finden würden - es wäre nicht auszudenken...

Der Initialisierungsteil von *Turbo3* bzw. *Graph3*

Mit diesen beiden Units verhält es sich ähnlich wie mit *Dos* und *Crt* - sie verwenden ein gemeinsames Segment und einen gemeinsamen Initialisierungsteil. Zusammen mit dem (von *Turbo3* und *Graph3* vorausgesetzten) Unit *Crt* lassen sich einige wundersame Kombinationen bilden:

- Ein Programm, das nur *Turbo3* mit **uses** aufführt, enthält ein eigenes Segment für *Crt* und ein eigenes Segment für *Turbo3*, die Initialisierungsteile beider Units werden mit voneinander getrennten CALL-Befehlen aufgerufen.
- Ein Programm, das nur *Graph3* mit **uses** aufführt, enthält ein gemeinsames Segment für *Crt* und *Graph3*, das den Namen GRAPH3 hat - und mit dem Initialisierungsteil von *Crt* beginnt.
- Ein Programm, das *Crt*, *Turbo3* und *Graph3* mit **uses** aufführt, enthält zwei Segmente für diese drei Units: *Crt* hat ein eigenes Segment und eine eigene Initialisierung, die auch separat aufgerufen wird; *Turbo3* und *Graph3* teilen sich ein Segment namens *Graph3*, das die Initialisierung von *Turbo3* enthält.

Lassen wir mögliche Kombinationen mit dem Unit *Dos* aus dem Spiel und kommen wir lieber gleich zur Initialisierung von *Turbo3*, die in Pascal geschrieben und wirklich übersichtlich ist: Sie besteht aus der Erzeugung der Textdatei-Variablen *Kbd* über einen Aufruf der (in *Turbo3* deklarierten) Prozedur *AssignKbd* und einer nachfolgenden Eröffnung dieser Datei über *Reset*:

```
TURBO3:@:
0109 89E5          MOV    BP,SP              ; V 5.0: PUSH BP / MOV BP,SP
010B BF0000        MOV    DI,0000            ; Adresse von KBD
010E 1E            PUSH   DS                   auf den Stack
010F 57            PUSH   DI
0110 0E            PUSH   CS                 ; plus CS für RETF
0111 E850FF        CALL   ASSIGNKBD          ; Initialisierung von KBD
```

AssignKbd ist eine in Pascal geschriebene Prozedur und setzt die Felder der übergebenen Variablen *Kbd* mit den folgenden Werten:

```
Handle    <undefiniert>   { durch Kbd vorgegeben }
Mode := FmClosed
BufSize := 128
Private   <undefiniert>
BufPos    <undefiniert>
BufEnd    <undefiniert>
BufPtr :=  <Adresse von Kbd + 128>
OpenFunc := TURBO3:0050
InOutFunc:= TURBO3:0000
FlushFunc:= TURBO3:0050
CloseFunc:= TURBO3:0050
UserData  <undefiniert>
Name := ''
```

Offensichtlich bestehen hier einige Unterschiede zu den Routinen *Assign* und *AssignCrt*: Zum einen bleibt eine ganze Reihe von Feldern undefiniert, zum anderen werden nicht nur *OpenFunc*, sondern auch die Zeiger auf die vier anderen Funktionen gesetzt.

Des Rätsels Lösung: Die Routine TURBO3:0050, auf die *OpenFunc*, *FlushFunc* und *CloseFunc* durch *AssignKbd* gesetzt werden, liefert schlicht das Funktionsergebnis 0 und zeigt damit eine »fehlerfreie Ausführung« an.

Die Routine TURBO3:0000 besteht aus einem Aufruf von *Crt.ReadKey*, wobei für gelesene NUL-Zeichen ein ESC (= `chr(27)`) eingesetzt wird, um mit der Version 3.0 von Turbo Pascal kompatibel zu bleiben.

Kbd ist zwar eine Textdatei-Variable und verfügt als solche über einen Puffer sowie die dazugehörigen Indexe bzw. Zeiger, arbeitet aber streng zeichenorientiert - der Puffer bleibt unbenutzt.

```
0114 BF0000        MOV   DI,0000
0117 1E            PUSH  DS                  ; Adresse von KBD erneut
0118 57            PUSH  DI                    auf den Stack
0119 9ADD02F63A    CALL  SYSTEM:RESET        ; RESET
011E 0E            PUSH  CS                  ; HIGHVIDEO arbeitet
011F E8C6FF        CALL  HIGHVIDEO             mit RET FAR
0122 89EC          MOV   SP,BP               ; V 5.0: MOV SP,BP / POP BP
0124 CB            RETF
```

Der zweite und letzte Teil der Initialisierung von *Turbo3*: Über einen Aufruf von *Reset* wird die Variable *Kbd* für Leseaktionen geöffnet, danach folgt ein Aufruf von (TURBO3:) *HighVideo*, der über eine direkte Zuweisung an die Variable *TextAttr* des Units *Crt* für folgende Zeichenausgaben »Gelb auf schwarzem Hintergrund« setzt. Das abschließende RETF führt zum Hauptprogramm des Pascal-Programms zurück.

Ein **Hinweis:** Spezielle Initialisierungen für Routinen von *Graph3* gibt es nicht.

Version 5.0

Außer den bereits im Listing angedeuteten PUSH- und POP-Befehlen für das BP-Register gibt es hier keine Unterschiede.

Typisierte und untypisierte Konstanten

Das Benutzerhandbuch beschränkt sich in Kapitel 25 auf die Feststellung, daß typisierte Konstanten nicht mehr wie bei der Version 3.0 im Codesegment, sondern im Datensegment gespeichert werden - *untypisierte* Konstanten sind nirgendwo erwähnt. Solange es sich bei letzteren um Integerwerte handelt, ist auch keine eigene Speicherung notwendig - der Compiler übersetzt Zuweisungen wie

```
IntVar := 4;
LongIntVar := $55790088;
```

direkt in Prozessorbefehle:

```
MOV Word Ptr [INTVAR], 0004
MOV Word Ptr [LONGINTVAR], 0088
MOV Word Ptr [LONGINTVAR+2], 5579
```

Wie es dagegen mit Konstanten aussieht, die sich nicht mit einem oder zwei Prozessorbefehlen bearbeiten lassen, zeigt das nächste Beispiel:

```
program TypedConst2;
const
  StrConst : string[20] = 'Typisierte Konstante';
begin
  Writeln(StrConst);
  Writeln('Untypisierter String');
end.
```

Dieses Programm wird von der Version 4.0 des Compilers folgendermaßen umgesetzt:

```
DATA:0000  14 54 79 70 69 73 69 65
           72 74 65 20 4B 6F 6E 73  <14> Typisierte Konstante
           74 61 6E 74 65
```

Die typisierte Konstante ist die erste Deklaration des Programms und wird deshalb im Datensegment ab der Adresse 0000 gespeichert.

```
CODE:0000  14 55 6E 74 79 70 69 73
           69 65 72 74 65 72 20 53  <14> Untypisierter String
           74 72 69 6E 67
```

Die untypisierte (d.h. direkt angegebene) Stringkonstante steht jeweils *direkt vor der Routine* im Codesegment, die diese Konstante benutzt - in unserem Beispiel also direkt vor dem Hauptprogramm. (Sie beginnt hier nur deshalb mit der Adresse 0000 des Segments, weil das Programm keine Prozeduren oder Funktionen deklariert). Die Startadresse des Hauptprogramms verschiebt sich entsprechend.

```
TYPEDCONST2:@:
** begin
0015 9A0000643A     CALL   SYSTEM:@        ; Initialisierung
001A 89E5           MOV    BP,SP           ; V 5.0: PUSH BP / MOV BP,SP
** Writeln(StrConst);
001C BF1601         MOV    DI,0116         ; Adresse von OUTPUT
001F 1E             PUSH   DS                auf den Stack
0020 57             PUSH   DI
0021 BF0000         MOV    DI,0000         ; Adresse der Konstanten
0024 1E             PUSH   DS                auf den Stack <- DS:0000
0025 57             PUSH   DI
0026 31C0           XOR    AX,AX           ; Formatlänge 0
0028 50             PUSH   AX
```

```
0029 9AF506643A     CALL   SYSTEM:06F5       ; Formatierung
002E 9A7906643A     CALL   SYSTEM:WRITELN    ; Ausgabe
0033 9A7702643A     CALL   SYSTEM:0277       ; Prüfung von IOResult
```

Die von *System* deklarierte Variable *Output* beginnt mit der Adresse $0116 des Datensegments, weil der Bereich DATA:$0000..15 von *StrConst* und der Bereich DATA:$0016..0115 von der Variablen *Input* belegt wird.

Die Ausgabe eines Strings über *Output* besteht aus vier Schritten:

1. Adresse von *Output* auf den Stack;
2. Adresse des Strings auf den Stack, die (nötigenfalls vom Compiler eingesetzte) Formatlänge dazu, Aufruf der Formatierungsroutine;
3. Aufruf von *Write* oder *Writeln*;
4. Prüfung von *IOResult*.

Wie das Listing des zweiten Aufrufs von *Writeln* zeigt, ist der Ablauf in beiden Fällen exakt derselbe.

```
** Writeln('Untypisierter String');
0038 BF1601         MOV    DI,0116          ; Adresse von OUTPUT
003B 1E             PUSH   DS                 auf den Stack
003C 57             PUSH   DI
003D BF0000         MOV    DI,0000          ; Adresse der Konstanten
0040 0E             PUSH   CS                 auf den Stack <- *CS*:0000
0041 57             PUSH   DI
0042 31C0           XOR    AX,AX            ; Formatlänge 0
0044 50             PUSH   AX
0045 9AF506643A     CALL   SYSTEM:06F5      ; Formatierung
004A 9A7906643A     CALL   SYSTEM:WRITELN   ; Ausgabe
004F 9A7702643A     CALL   SYSTEM:0277      ; Prüfung von IOResult
** end.
0054 89EC           MOV    SP,BP            ; V 5.0: MOV SP,BP / POP BP
0056 31C0           XOR    AX,AX            ; Exitcode 0
0058 9AF301643A     CALL   SYSTEM:HALT      ; Programmende
```

Dasselbe Verfahren wird für untypisierte Konstanten der Typen *Single..Comp* verwendet. Der Compiler merkt erfreulicherweise ab der Version 4.0 anhand des »Zuweisungsziels«, wann eine Konstante wie 2 als Integer und wann sie als Fließkommazahl gespeichert werden muß. Ein Beispiel dazu:

```
program UntypedReals;
{$N+}
var
  R: Extended;   X: Integer;

procedure proc1;
{ Nur deklariert, damit die untypisierten Konstanten nicht schon
  wieder ab der Adresse 0 im Codesegment stehen }
begin end;
```

```
procedure proc2;
begin
  R := 2;   { wird als 0.2 E+01 im Codesegment gespeichert }
  X := 2;   { direkte Zuordnung }
end;

begin
  proc1;   { Der Aufruf ist notwendig - sonst wirft der Linker }
  proc2;         {  die Prozedur ganz hinaus. }
end.
```

Die Umsetzung dieses Programms durch den Compiler ist hier mehr oder weniger unkommentiert wiedergegeben - hauptsächlich deshalb, weil sich das nächste Kapitel auf rund 50 Seiten mit dem Thema »Coprozessor« auseinandersetzt.

```
UNTYPEDREALS:PROC1:
0000 55              PUSH   BP                     ; begin
0001 89E5            MOV    BP,SP
0003 89EC            MOV    SP,BP                  ; end
0005 5D              POP    BP
0006 C3              RET

0007  00 00 00 00 00 00 00 80 00 40  +0.2E+1       ; = 2.0
UNTYPEDREALS:PROC2:
0011 55              PUSH   BP                     ; begin
0012 89E5            MOV    BP,SP
0014 9B              WAIT                          ;   R := 2.0;
0015 2EDB2E0700      FLD    TByte Ptr CS:[0007]    ; Laden der Konstanten
001A 9B              WAIT
001B DB3E0000        FSTP   TByte Ptr [R]          ; Speicherung in R
001F 9B              WAIT
0020 C7060A000200    MOV    Word Ptr [X],0002      ;   X := 2;
0026 89EC            MOV    SP,BP                  ; end;
0028 5D              POP    BP
0029 C3              RET
UNTYPEDREALS:@:
002A 9A0000693A      CALL   SYSTEM:@               ; Initialisierung
002F 9AAF02693A      CALL   SYSTEM:02AF            ; Coprozessor-Init
0034 89E5            MOV    BP,SP                  ; begin
0036 E8C7FF          CALL   PROC1                  ;   proc1;
0039 E8D5FF          CALL   PROC2                  ;   proc2;
003C 89EC            MOV    SP,BP                  ; end;
003E 31C0            XOR    AX,AX                  ; Exitcode 0
0040 9AF301693A      CALL   SYSTEM:HALT            ; Programmende
```

Version 3.0

In der Version 3.0 werden sowohl typisierte als auch untypisierte Konstanten im Codesegment gespeichert - und zwar mitten im Programmcode. Das Programm

TypedConst2 wird (ohne jede Veränderung des Quelltextes) von der Version 3.01A folgendermaßen umgesetzt:

```
2D9C E91500        JMP     2DB4            ; Sprung um den String herum
2D9F 14 54 79 70 69 73 69 65 72 74 65 20 4B 6F 6E 73        <14> Typisierte
     74 61 6E 74 65                                              Konstante
2DB4 E8A1F7        CALL    2558            ; SetStdOut
2DB7 BF9F2D        MOV     DI,2D9F         ; Adresse der Konstanten
2DBA 0E            PUSH    CS                auf den Stack
2DBB E80EE4        CALL    11CC            ; StrToStack
2DBE B80000        MOV     AX,0000         ; Formatlänge 0
2DC1 E862FB        CALL    2926            ; WriteString
2DC4 E8AEFB        CALL    2975            ; OutCRLF
2DC7 E858E2        CALL    1022            ; CheckIOResult
```

Wenn man davon absieht, daß der Prozessor um die typisierte Konstante herumspringen muß, verläuft die Ausgabe dieses Strings in recht geordneten Bahnen und vor allem sehr effizient: Der Aufruf von *SetStdOut* stellt das Äquivalent zur Übergabe von *Output* dar, danach wird die Konstante auf den Stack kopiert und zusammen mit der Formatlänge an *Write* übergeben. Die Versionen ab 4.0 verwenden getrennte Aufrufe für Formatierung und Ausgabe, in der Version 3.0 erledigt *Write* die Formatierung gleich mit. Über den Aufruf von *OutCRLF* wird die Ausgabe mit einem Zeilenvorschub abgeschlossen, bevor es mit der Prüfung von *IOResult* weitergeht (auch hier war der Schalter {$I+} während der Compilierung gesetzt).

```
2DCA E88BF7        CALL    2558            ; SetStdOut
2DCD E88DFB        CALL    295D            ; WriteStrLit
2DD0  14 55 6E 74 79 70 69 73 69 65 72 74 65 72 20 53   <14> Untypisierter
      74 72 69 6E 67                                          String
2DE5 E88DFB        CALL    2975            ; OutCRLF
2DE8 E837E2        CALL    1022            ; CheckIOResult
2DEB E90000        JMP     2DEE
2DEE 33C0          XOR     AX,AX           ; Exitcode 0
2DF0 E896DE        CALL    0C89            ; Programmende
```

Je effizienter, desto abenteuerlicher: *WriteStrLit* ermittelt die Adresse des Strings aus der Rücksprungadresse des aufrufenden Programms, gibt den String aus und *erhöht die Rücksprungadresse während der Ausgabe entsprechend* - danach geht es im Hauptprogramm tatsächlich mit dem Aufruf von *OutCRLF* weiter. Formal gesehen handelt es sich bei dieser Art von »Übergabe« um einen Pseudo-**var**-Parameter: *WriteStrLit* könnte den String verändern - weil der Compiler aber genau weiß, daß die Routine es nicht tut, spart er sich ein Kopieren auf den Stack.

Der Vergleich des Programms *UntypedReals* erfordert einige Modifikationen des Quelltextes - nicht zuletzt deshalb, weil die Version 3.0 nur den Typ *Real* kennt. (Warum dann die erste Version dieses Beispiels unbedingt mit *Extended* arbeiten muß? Weil sich die Versionen 4.0 und 5.0 im Modus {$N+} mit *Real* ausgesprochen schwertun - siehe Kapitel 2).

```
program UntypedReals;
{$K- keine Stackprüfung, bei 4.0 und 5.0 die Standardvorgabe }

var
  R1,R2: Real;
  X: Integer;

procedure proc2;
begin
  R1 := 2;      { wird als 2 gespeichert und in Real umgewandelt }
  R2 := 2.0;    { wird als 0.2 E+0 im Codesegment gespeichert }
  X := 2;       { direkte Zuordnung }
end;

begin
  proc2;
end.
```

Wie das folgende Listing von *Proc2* zeigt, speichert die Version 3.0 untypisierte *Real*-Konstanten ebenfalls mitten im Programmcode - eine zusätzliche »Beweisroutine« wie *Proc1* ist in diesem Beispiel also unnötig.

```
UNTYPEDREALS:PROC2:
2D9F 55          PUSH    BP
2DA0 8BEC        MOV     BP,SP
2DA2 55          PUSH    BP
2DA3 E90000      JMP     2DA6
2DA6 BF6002      MOV     DI,0260
2DA9 1E          PUSH    DS                  ; Adresse von R1 auf
2DAA 57          PUSH    DI                    den Stack
2DAB B80200      MOV     AX,0002             ; Integerwert 2
2DAE E87EED      CALL    1B2F                ; IntRealArg
2DB1 E8DFEB      CALL    1993                ; StackToRealVar
```

Der Compiler der Version 3.0 orientiert sich bei der Speicherung numerischer Konstanten ausschließlich an der Konstanten selbst und läßt die jeweilige »Zielvariable« unberücksichtigt. Weil Pascal (im Gegensatz zu C) aber Zuweisungen der Form *Real_Var := Int_Konstante* erlaubt, enthält die Laufzeitbibliothek eine Spezialroutine (*IntRealArg*), die einen Integerwert im AX-Register in das Format *Real* umwandelt und danach auf den Stack packt. (Daher der Name - es gibt auch noch eine »echte« Umrechnungsroutine (*IntToReal*), die ihr Ergebnis in den Registern AX/BX/CX zurückliefert, aber nur bibliotheksintern verwendet wird). Der Aufruf von *StackToRealVar* schreibt den von *IntRealArg* erzeugten Wert danach in die Variable *R1*.

```
2DB4 BF6602      MOV     DI,0266
2DB7 1E          PUSH    DS                  ; Adresse von R2 auf
2DB8 57          PUSH    DI                    den Stack
```

```
2DB9 E8C6EB        CALL    1982                ; CSegRealToStack
2DBC 82 00 00 00 00 00             ; 0.2 E+01
2DC2 E8CEEB        CALL    1993                ; StackToRealVar
```

CSegRealToStack verwendet dasselbe abenteuerliche Verfahren wie *WriteStrLit*: Das Argument wird über die RET-Adresse zur aufrufenden Routine gelesen, die dabei entsprechend erhöht wird. Der Aufruf von *StackToRealVar* speichert den Wert in der Variablen *R2*.

```
2DC5 B80200        MOV     AX,0002
2DC8 A36C02        MOV     [X],AX              ; X := 2;
2DCB E90000        JMP     2DCE                ; end;
2DCE 8BE5          MOV     SP,BP
2DD0 5D            POP     BP
2DD1 C3            RET
```

Die Zuordnung von Integerwerten findet auch in der Version 3.0 direkt statt - wenn man einmal davon absieht, daß der Compiler die Möglichkeiten des 8086 zum direkten Setzen von Speicherzellen (`MOV Word Ptr [X], 0002`) außer acht läßt und statt dessen das Register AX benutzt.

Version 5.0

Bis auf die wohl unvermeidlichen PUSH- und POP-Befehle für das BP-Register, eine andere Adresse für die 8087-Initialisierungsroutine (SYSTEM:0761) und schließlich die Tatsache, daß die ersten $003C Bytes des Datensegments mit typisierten Konstanten von *System* belegt sind (vgl. »Units, Variablen und Konstanten«), zeigt ein Listing mit DEBUG oder SYMDEB keinen Unterschied - allerdings erst *nach* Ausführung des Programms auf einem System, das auch wirklich mit einem Coprozessor ausgerüstet ist. *Vor* der Ausführung von *UntypedReals* zeigt sich die Prozedur *Proc2* für DEBUG als Folge scheinbar völlig sinnloser Werte; derselbe Versuch mit SYMDEB ergibt:

```
UNTYPEDREALS:PROC2:
0011 55            PUSH    BP                  ; begin
0012 89E5          MOV     BP,SP

0014 CD3C9B2E0700  INT     3C ;FLD   TByte Ptr CS:[0007]  ; R := 2;
001A CD373E3C00    INT     37 ;FSTP  TByte Ptr [R]
001F CD3D          INT     3D ;FWAIT

0021 C70646000200  MOV     Word Ptr [X],0002     ; X := 2;
0027 89EC          MOV     SP,BP                 ; end
0029 5D            POP     BP
002A C3            RET
```

Für diese Merkwürdigkeiten ist der 8087-Emulator der Version 5.0 verantwortlich, um den man übrigens auch im Modus {$N+,E-} nicht herumkommt*).

Prozeduren und Funktionen

Der Aufruf einer Prozedur ohne Parameter stellt den einfachsten Fall dar und ist so schnell besprochen, daß er nicht einmal eine eigene Überschrift bekommt:

- Wenn die Prozedur im Modul des Hauptprogramms deklariert wurde, erfolgt der Aufruf mit CALL und der Rücksprung mit RET. Dasselbe gilt für lokale Deklarationen, d.h. Prozeduren, die innerhalb einer anderen Prozedur oder Funktion deklariert wurden.
- Wenn die Prozedur in einem anderen Unit deklariert wurde, erfolgt der Aufruf mit CALL FAR und der Rücksprung mit RETF. Dasselbe gilt für alle Prozeduren, die mit {$F+} und {$F-} eingeklammert wurden und nicht lokal zu einer anderen Routine deklariert sind.
- Wenn die Prozedur im Interfaceteil eines Units deklariert wurde und vom Implementationsteil dieses Units aus aufgerufen wird, spart der Compiler einige Mikrosekunden Laufzeit, indem er zuerst das CS-Register auf den Stack bringt und dann einen normalen CALL-Befehl verwendet.

Parameterlose Funktionen

Der Unterschied zwischen einer Prozedur und einer Funktion besteht darin, daß letztere ein Ergebnis zurückliefert. Nein - dieses Statement wendet sich nicht an die Zielgruppe der (hoffentlich) fiktiven Buchreihe »Computern leichtgemacht«: Weil Pascal nicht vorschreibt, *wann* die Zuweisung des Funktionsergebnisses erfolgen soll, muß dafür ein temporärer Speicherplatz reserviert werden.

Deshalb bleibt uns die Auseinandersetzung mit lokalen Variablen, Stackmanipulationen und Indizierungen über das BP-Register bereits an dieser Stelle nicht erspart. Ein Beispiel:

```
program FuncDemo1;
var  A: Integer;

function IntFunc: Integer;
begin
  IntFunc := 2;
           { <-- hier könnte eine beliebige Anzahl weiterer }
end;             {  Anweisungen folgen }
```

*) Wir lassen es vorläufig mit dieser dunklen Andeutung bewenden - die dazugehörigen Hintergründe finden sich in Kapitel 2 bzw. Anhang C.

```
begin
  A := IntFunc;
end.
```

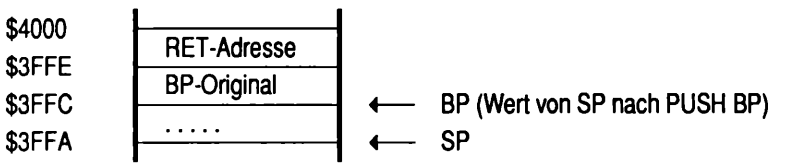
Die Umsetzung dieses Programms durch den Compiler:

```
FUNCDEMO1:INTFUNC:
0000 55            PUSH   BP
0001 89E5          MOV    BP,SP
0003 83EC02        SUB    SP,+02                ; Platz fürs Ergebnis
0006 C746FE0200    MOV    Word Ptr [BP-02],0002 ; INTFUNC := 2;
000B 8B46FE        MOV    AX,[BP-02]            ; Ergebnis -> AX
000E 89EC          MOV    SP,BP                 ; Stack wieder zurück
0010 5D            POP    BP
0011 C3            RET

FUNCDEMO1:@:
0012 9A0000623A    CALL   SYSTEM:@              ; Initialisierung
0017 89E5          MOV    BP,SP                 ; V 5.0: PUSH BP / MOV BP,SP
0019 E8E4FF        CALL   INTFUNC               ; A := INTFUNC
001C A30000        MOV    [A],AX
001F 89EC          MOV    SP,BP                 ; V 5.0: MOV BP,SP / POP BP
0021 31C0          XOR    AX,AX                 ; Exitcode 0
0023 9AF301623A    CALL   SYSTEM:HALT           ; Programmende
```

Wenn wir von der Standardgröße des Stacks ausgehen (d.h. beim Programmstart den Wert $4000 für SP annehmen), dann laufen die diversen Stackoperationen folgendermaßen ab:

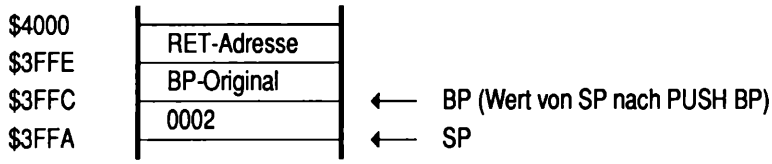
- Aufruf von *IntFunc*: Die Rücksprungadresse zum Hauptprogramm wird auf den Stack gebracht.

- Die ersten drei Befehle innerhalb von *IntFunc* bringen das BP-Register auf den Stack, weisen BP den neuen Wert von SP zu und erniedrigen SP danach um zwei Bytes, d.h. schaffen Platz für das Funktionsergebnis.

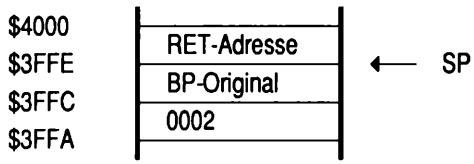


- Die nächsten beiden Befehle machen sich eine spezielle Eigenschaft des BP-Registers zunutze: Bei Indizierungen über dieses Register stellt der 8086 implizit das Präfix `SS:` voran, adressiert also das Stacksegment. BP hat momentan

den Wert $3FFC - mit `Word Ptr [BP-02]` werden folglich die Speicherzellen SS:$3FFA/B angesprochen.



- Mit `MOV SP,BP` wird das SP-Register auf den Wert $3FFC zurückgesetzt, der folgende POP-Befehl stellt den ursprünglichen Inhalt von BP wieder her und erhöht SP auf $3FFE.



- Der Rücksprung zum Hauptprogramm liest die obersten zwei Bytes des Stacks und erhöht SP wieder auf den Anfangswert ($4000). Es sollte noch einmal betont werden, daß dadurch keine Veränderung der Werte stattfindet, die auf dem Stack gespeichert sind - erst durch den Aufruf von *Halt* wird der Bereich $3FFC..$3FFF des Stacksegments mit einer neuen (und nicht mehr verwendeten) Rücksprungadresse überschrieben.

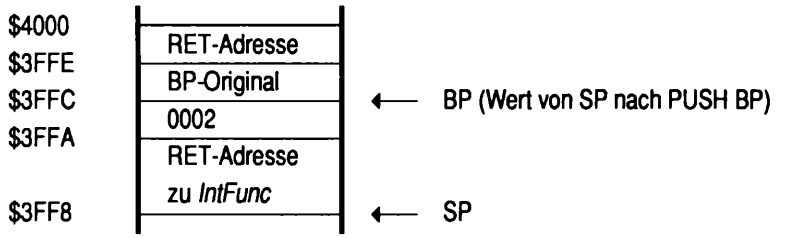
Wieso wird SP innerhalb von *IntFunc* verändert, obwohl die Adressierung des Funktionsergebnisses über das BP-Register geschieht? Weil sich der Stack dann auch in *IntFunc* auf völlig normale Weise zur Speicherung von Zwischenergebnissen und beim Aufruf weiterer Routinen verwenden läßt. In einer erweiterten Version von *IntFunc* wie

```
function IntFunc: Integer;
begin
  IntFunc := 2;
  ProcX;
end;
```

wird der Aufruf der Prozedur *ProcX* einfach vom Compiler als CALL-Befehl eingeschoben:

```
     ....
0003 83EC02        SUB    SP,+02                ; Platz fürs Ergebnis
0006 C746FE0200    MOV    Word Ptr [BP-02],0002 ; INTFUNC := 2;
000B E8xxxx        CALL   PROCX
000E 8B46FE        MOV    AX,[BP-02]            ; Ergebnis -> AX
0011 89EC          MOV    SP,BP                 ; Stack wieder zurück
     ....
```

Vor der Ausführung des ersten Befehls innerhalb von *ProcX* sieht der Stack so aus:



Damit dürfte auch klar sein, warum jede Routine das BP-Register vor einer Veränderung zwischenspeichern muß: *IntFunc* verläßt sich darauf, daß der Inhalt dieses Registers konstant bleibt und verwendet es nach dem Aufruf nicht nur zum Lesen des Funktionsergebnisses (und zur Adressierung eventueller weiterer lokaler Variablen), sondern auch zum Zurücksetzen des Stacks. Deshalb kann das folgende »Achtung« eigentlich gar nicht groß genug ausfallen:

ACHTUNG: Prozeduren und Funktionen, die den Wert des BP-Registers versehentlich verändern, führen mit *absoluter Sicherheit* zum Absturz des Systems. (Die Idee, das BP-Register doch einfach innerhalb der *aufrufenden* Routine zu speichern, bringt außer weiteren Komplikationen bei der Übergabe von Parametern leider auch nicht viel mehr als längere und langsamere Programme - wir müssen auch in der *aufgerufenen* Routine auf irgendeine Weise dafür sorgen, daß SP vor dem Rücksprung wieder auf den richtigen Wert gesetzt wird).

Version 3.0

Die einzige Gemeinsamkeit mit den Versionen 4.0 und 5.0 besteht darin, daß Funktionsergebnisse des Typs *Integer* ebenfalls im Register AX zurückgeliefert werden. Ansonsten läuft hier wirklich alles anders, weil die Version 3.0 für lokale Variablen und Funktionsergebnisse unterschiedliche Speicherkonzepte verwendet. Das Programm *FuncDemo1* wird von der Version 3.01A des Compilers (bei abgeschalteter Stackprüfung) folgendermaßen umgesetzt:

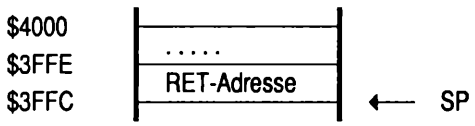
```
2D9C E91900        JMP     2DB8                ; Sprung zum Hauptprogramm

FUNCDEMO1:INTFUNC:
2D9F 55            PUSH    BP
2DA0 8BEC          MOV     BP,SP
2DA2 55            PUSH    BP
2DA3 E90000        JMP     2DA6
2DA6 B80200        MOV     AX,0002
2DA9 894604        MOV     [BP+04],AX          ; INTFUNC := 2;
2DAC E90000        JMP     2DAF
2DAF 8B4604        MOV     AX,[BP+04]          ; Ergebnis -> AX
2DB2 8BE5          MOV     SP,BP
2DB4 5D            POP     BP
2DB5 C20200        RET     0002                ; Stack zurück
```

```
FUNCDEMO1:@
2DB8 4C          DEC     SP                  ; Platz für das Funktions-
2DB9 4C          DEC     SP                    ergebnis auf dem Stack
2DBA E8E2FF      CALL    INTFUNC             ; Aufruf
2DBD A36002      MOV     [A],AX              ; Ergebnis -> Variable A
2DC0 E90000      JMP     2DC3

2DC3 33C0        XOR     AX,AX               ; Exitcode 0
2DC5 E8C1DE      CALL    0C89                ; Programmende via HALT
```

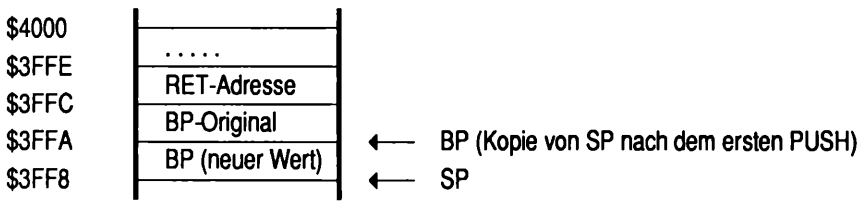
Das Reservieren von Speicherplatz für Funktionsergebnisse findet hier offensichtlich innerhalb des *aufrufenden* Programms statt - die Freigabe des reservierten Bereichs dagegen am Ende der *aufgerufenen* Routine*).

Wenn wir auch hier davon ausgehen, daß SP zu Beginn des Programms den Wert $4000 hat, dann können wir einen ungefähren Vergleich versuchen:

- Die ersten beiden Befehle des Hauptprogramms erniedrigen SP um zwei, darauf folgt der Aufruf von *IntFunc*. Der Zustand des Stacks vor der Ausführung des ersten Befehls dieser Funktion:

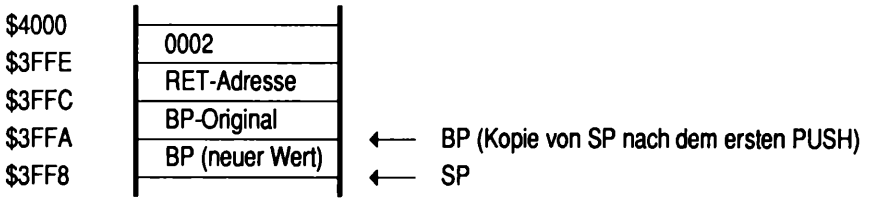


- Die ersten drei Befehle von *IntFunc* bringen das BP-Register auf den Stack (d.h. erniedrigen SP um zwei), weisen BP den jetzigen Stand von SP zu und bringen diesen Wert erneut auf den Stack.

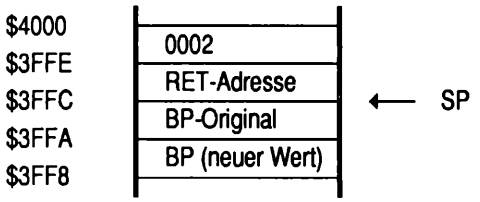


- Das Funktionsergebnis wird wie in den Versionen 4.0 und 5.0 mit einer Indizierung über das BP-Register adressiert - nur daß diese lokale Variable hier oberhalb der Rücksprungadresse liegt. BP hat momentan den Wert $3FFA, mit `Word Ptr [BP+4]` werden also die Speicherzellen SS:$3FFE/F angesprochen.

*) Letzteres nur dann, wenn die Funktion ihr Ergebnis nicht über den Stack zurückliefert: Ergebnisse der Typen *Real* und *String* werden von der aufrufenden Routine bei der Speicherung in die Zielvariable der Zuweisung vom Stack geholt.



- Der Befehl `MOV SP,BP` setzt SP auf den Wert $3FFA zurück, das anschließende `POP BP` stellt den Originalwert von SP wieder her und erhöht SP auf $3FFC.



- Der Befehl `RET 0002` liest die Rücksprungadresse vom Stack, erhöht SP dabei um 2 und addiert dann noch einmal den Wert 2 zu SP hinzu. Bei der Ausführung des nächsten Befehls im Hauptprogramm hat SP also wieder seinen ursprünglichen Wert ($4000).

Wie sich mit der Erweiterung von *IntFunc* um einen zusätzlichen Funktionsaufruf zeigen läßt, wird dieses Konzept konsequent durchgehalten, d.h. die Reservierung von Speicherplatz für Funktionsergebnisse ist immer Sache des aufrufenden Programms:

```
function IntFunc: Integer;
begin
  IntFunc := 2;
  IntFunc := FuncX;    { Ergebnistyp Integer }
end;
```

Der Aufruf von *FuncX* in dieser Version von *IntFunc*:

```
    .....
2DA6 B80200        MOV     AX,0002
2DA9 894604        MOV     [BP+04],AX          ; INTFUNC := 2;
2DAC 4C            DEC     SP                  ; Platz für das
2DAD 4C            DEC     SP                    Ergebnis von FUNCX
2DAE E8xxxx        CALL    FUNCX               ; Aufruf
2DB1 894604        MOV     [BP+04],AX          ; Ergebnis -> INTFUNC
2DB4 E90000        JMP     2DB7
2DB7 8B4604        MOV     AX,[BP+04]          ; INTFUNC -> AX
2DBA 8BE5          MOV     SP,BP
    ....
```

So durcheinander der erzeugte Code auch aussieht - das Reservieren von Speicherplatz *oberhalb* der Rücksprungadresse ist einer der vielen Geniestreiche, mit denen die Programmierer von Borland das gesamte Programmierpaket in seine unglaublich geringe Größe gepreßt haben: Der Compiler muß hier bei der Bearbeitung lokaler Variablen nicht zwischen Funktionen und Prozeduren unterscheiden. (Was den Sinn dieser »Erklärung« angeht - siehe »Unterschiede zwischen Funktionen und Prozeduren«, Seite 74).

Stringfunktionen

Ein Zitat aus Kapitel 25 des Benutzerhandbuchs: »Für String-Typen muß die aufrufende Funktion vor den eigentlichen Parametern einen Zeiger auf einen temporären Speicherbereich auf den Stack legen. Die Funktion liefert ihr Ergebnis in diesem Bereich zurück und darf den Zeiger (im Gegensatz zu allen anderen Parametern) nicht vom Stack entfernen.«

Die beiden dort gezeigten Beispiele *UpperCase* und *StringOf* zeigen sogar detailliert, wie man eine solche Funktion zusammenbaut - die Frage aber, wie der Compiler einen solchen »temporären Speicherbereich« erzeugt, bleibt unbeantwortet. Aus zwei guten Gründen: Erstens ist die Sache nicht gerade einfach, zweitens hätte man an dieser Stelle wirklich eine Optimierung einbauen sollen - und das haben die Programmierer von Borland nicht getan.

Ein Beispielprogramm, das auch gleich die fehlende Optimierung demonstrieren soll:

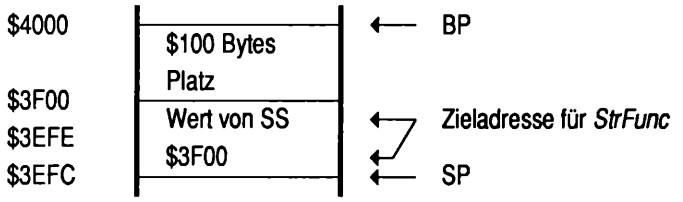
```
program StrFunc1;
type
 Str4 = string[4];     { ein String mit maximal 4 Zeichen }
var
 s: Str4;

function StrFunc: Str4;
begin
  StrFunc := '****';
end;

begin
  s := StrFunc;
end.
```

Und das Ergebnis der Compilierung, wobei wir diesmal mit dem Hauptprogramm anfangen:

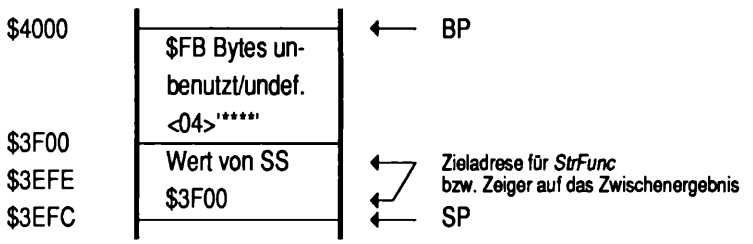
```
STRFUNC1:@:
** begin
001F 9A0000653A     CALL   SYSTEM:@
0024 89E5           MOV    BP,SP                  ; V 5.0: PUSH BP / MOV BP,SP
0026 81EC0001       SUB    SP,0100                ; Platz fürs Ergebnis (!)
```

```
** s := StrFunc;
002A 8DBE00FF        LEA    DI,[BP+FF00]          ; = "MOV DI, BP-$100"
002E 16              PUSH   SS                    ; Zieladresse als Parameter
002F 57              PUSH   DI                      auf den Stack
0030 E8D2FF          CALL   STRFUNC               ; Aufruf
```

Hier haben wir endlich den Grund für die Befehle `MOV BP,SP` und `MOV SP,BP` am Anfang und Ende jedes Hauptprogramms: Das Reservieren des »temporären Speicherbereichs« geschieht auf dieselbe Weise wie innerhalb einer Prozedur oder Funktion, nämlich über das Festhalten der momentanen Spitze des Stacks in BP und eine entsprechende Erniedrigung von SP. Wenn wir von der Standardvorgabe {$M 16384,...} ausgehen, sieht der Stack direkt vor dem Aufruf von *StrFunc* so aus:



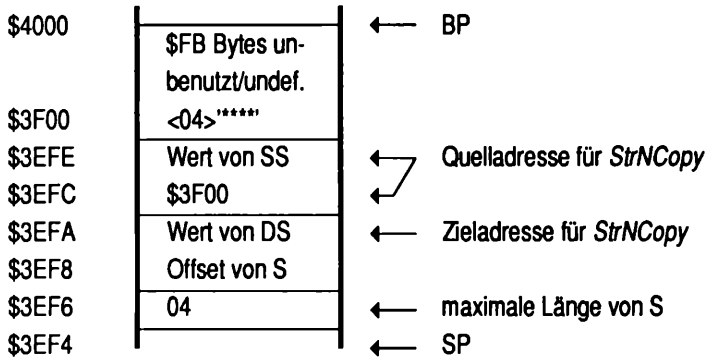
Die Routine *StrFunc* bekommt also einen Zeiger übergeben, der auf die Adresse $3F00 innerhalb des Stacksegments zeigt. In diesen Bereich schreibt sie ihr Funktionsergebnis, wobei der Zeiger (streng nach Vorschrift) ebenfalls auf dem Stack bleibt. Nach dem Rücksprung zum Hauptprogramm sieht der Stack also so aus:



Der größte Teil der reservierten $100 Bytes ist nicht belegt - schließlich liefert *StrFunc* auch nur einen String mit maximal 4 Zeichen Länge als Ergebnis. Die nächsten Befehle des Hauptprogramms stellen den zweiten Teil der Zuweisung dar:

```
0033 BF0000          MOV    DI,0000
0036 1E              PUSH   DS                    ; Adresse der Variablen
0037 57              PUSH   DI                      S auf den Stack
0038 B80400          MOV    AX,0004               ; Maximal 4 Zeichen dürfen
003B 50              PUSH   AX                      kopiert werden
003C 9A8702653A      CALL   SYSTEM:STRNCOPY       ; Kopie -> Variable S
```

Zusätzlich zu dem Zeiger auf den temporären Speicherbereich wird die Adresse der Variablen S auf den Stack gebracht, darunter kommt die Maximallänge von S. Der Stack vor dem Aufruf der Routine *StrNCopy*:



StrNCopy ist eine interne Routine von *System*, die für beliebige Zuweisungen der Form *Str_Var := Stringausdruck* verwendet wird. Ihr Aufruf kopiert hier maximal 4 Zeichen aus dem temporären Speicherbereich in die Variable S, danach werden die übergebenen Parameter vom Stack entfernt: SP hat nach dem Rücksprung ins Hauptprogramm wieder den Wert $3F00.

```
** end.
0041 89EC          MOV    SP,BP              ; V 5.0: MOV SP,BP / POP BP
0043 31C0          XOR    AX,AX              ; Exitcode 0
0045 9AF301653A    CALL   SYSTEM:HALT        ; Programmende
```

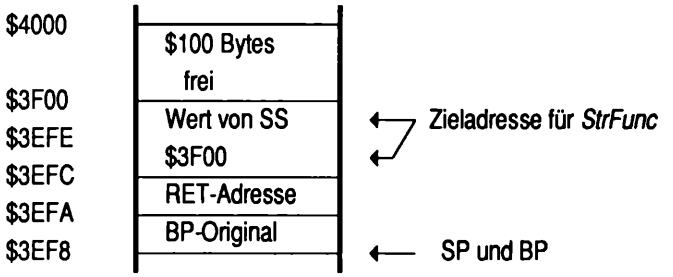
Unabhängig von weiteren Operationen bleibt SP innerhalb des Hauptprogramms auf dem Wert $3F00 - der »temporäre« Bereich wird also erst zum Programmende wieder freigegeben.

Der Rest des Programms: Vor der Funktion *StrFunc* befindet sich die untypisierte Stringkonstante, die innerhalb der Funktion verwendet wird*).

```
0000  04 2A 2A 2A 2A          <04> ****    (untypisierte Stringkonstante)

STRFUNC1:STRFUNC:
** begin
0005 55            PUSH   BP
0006 89E5          MOV    BP,SP
```

*) Wie im Abschnitt »Typisierte und untypisierte Konstanten« (Seite 43) demonstriert, ist diese Stringkonstante nur deshalb ab der Adresse 0000 des Codesegments gespeichert, weil *StrFunc* die erste Routine des Programms darstellt.

Der Stack sieht nach diesen beiden Befehlen so aus:



Die Routine »weiß«, daß sie als *near* (mit zwei Bytes Rücksprungadresse) aufgerufen worden ist - zwischen der Adresse, auf die BP momentan zeigt, und der übergebenen Zieladresse befinden sich insgesamt 4 Bytes auf dem Stack.

```
** StrFunc := '****';
0008 BF0000        MOV   DI,0000
000B 0E            PUSH  CS                  ; Adresse der String-
000C 57            PUSH  DI                    konstanten auf den Stack
000D C47E04        LES   DI,[BP+04]          ; Zieladresse -> ES:DI
0010 06            PUSH  ES                  ; Zieladresse auf
0011 57            PUSH  DI                    den Stack
0012 B80400        MOV   AX,0004             ; maximal 4 Zeichen
0015 50            PUSH  AX                    dürfen kopiert werden
0016 9A8702653A    CALL  SYSTEM:STRNCOPY     ; Kopie -> Stack
** end;
001B 89EC          MOV   SP,BP
001D 5D            POP   BP
001E C3            RET
```

Dieser Aufruf von *StrNCopy* kopiert die typisierte Stringkonstante (CS:0000) in den temporären Speicherbereich, den das Hauptprogramm zur Verfügung gestellt hat (SS:$3F00). Da *StrNCopy* übergebene Parameter vom Stack entfernt, muß die Zieladresse zuvor ein zweites Mal auf den Stack gebracht werden.

Nun gut - das Hauptprogramm reserviert also $100 Bytes »temporär« auf dem Stack und gibt sie erst beim Ende des Programms wieder frei. Außerdem läßt sich durch eine Erweiterung des Programms zeigen, daß mehrere Aufrufe von *StrFunc* oder anderer Funktionen *denselben temporären Speicherplatz benutzen**) - BP und SP werden nämlich auch innerhalb des Hauptprogramms nicht weiter verändert. Wo ist der Ansatzpunkt für die so dringend benötigte Optimierung?

Er liegt in der Stringlänge begraben: Obwohl wir sämtliche verwendeten Strings und Ergebnistypen explizit auf eine Länge von vier Zeichen begrenzt haben, werden $100 Bytes durch den Compiler reserviert. Ärgerlich daran: derselbe Mecha-

*) Verschachtelte Aufrufe von Stringfunktionen wie `s := Copy(x,1,4) + '***'` reservieren entsprechend mehr Speicherplatz ($100 Bytes für jedes Zwischenergebnis).

nismus wird nicht nur innerhalb des Hauptprogramms, sondern auch bei Prozeduren und Funktionen für Strings verwendet. Ein Beispiel:

```
type Str1: string[1];

function KurzStr: Str1;
begin
  KurzStr:= 'a';
end;

procedure x;
var
  XStr: Str1;
begin
  XStr := KurzStr;
end;
```

Für die Variable *XStr* werden tatsächlich nur zwei Bytes auf dem Stack belegt - da die Prozedur *x* aber eine Stringfunktion aufruft, braucht sie insgesamt $102 Bytes für lokale Variablen. Allgemein:

- Jede Routine, die irgendeine Stringfunktion aufruft - egal, ob *Copy*, *Concat*, *Insert* usw. oder selbstdefiniert - belegt mindestens $100 Bytes auf dem Stack.
- Die Deklaration eines »kürzeren« Stringtyps als Ergebnis einer Funktion ist sinnlos. Sie wird de facto vom Compiler ignoriert.

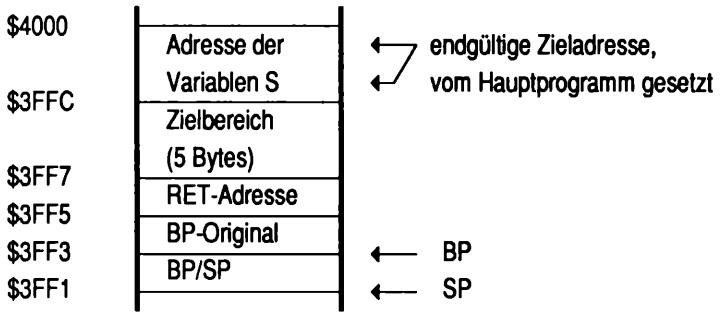
Version 3.0

Da die Versionen 4.0 und 5.0 ebenfalls vor dem Funktionsaufruf Platz für das Ergebnis auf dem Stack reservieren, sind die Ähnlichkeiten hier wesentlich größer als bei Funktionen mit numerischem Ergebnis.

Achtung! Stringfunktionen »wissen« in der Version 3.0, daß der temporäre Bereich direkt oberhalb der RET-Adresse auf dem Stack liegt, und setzen diese Adresse zusammen mit SS selbständig ein - eine direkte Zeigerübergabe gibt es also nicht.

```
2D9C E92700        JMP     2DC6                ; Sprung zum Hauptprogramm

STRFUNC1:STRFUNC:
2D9F 55            PUSH    BP
2DA0 8BEC          MOV     BP,SP
2DA2 55            PUSH    BP
2DA3 E90000        JMP     2DA6
```

Der Stack nach der Ausführung dieser Befehle (wobei der Anfangswert von SP wieder mit $4000 angenommen wurde):



Der Aufruf von *CSegStrToStack* kopiert die Stringkonstante aus dem Codesegment auf den Stack (d.h. in den Bereich SS:$3FEC..F0). SP hat danach den Wert $3EFC.

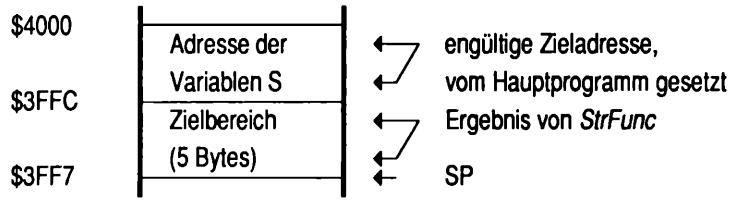
```
2DA6 8D7E04        LEA     DI,[BP+04]          ; Zieladresse (direkt
2DA9 16            PUSH    SS                    oberhalb RET-Adresse)
2DAA 57            PUSH    DI                    auf den Stack
2DAB E837E4        CALL    11E5                ; CSegStrToStack
2DAE 04 2A 2A 2A 2A                  ; <04> ****
2DB3 B104          MOV     CL,04               ; maximal 4 Zeichen
2DB5 E843E4        CALL    11FB                ; StackToStr -> Stack(!)
2DB8 E90000        JMP     2DBB
2DBB BA0000        MOV     DX,0000             ; Begrenzung des
2DBE B104          MOV     CL,04                 Endergebnisses
2DC0 8BE5          MOV     SP,BP                 auf 4 Zeichen
2DC2 5D            POP     BP
2DC3 E9D2E6        JMP     1498                ; RET via PopStrDiff
```

Eine Längenbegrenzung findet durch *CSegStrToStack* nicht statt. Der Aufruf von *StackToStr* kopiert maximal 4 Zeichen vom Stack in den Zielbereich (ab SS:$3FF7), SP wird um die Anzahl der kopierten Zeichen erhöht.

Der Aufruf von *PopStrDiff* am Ende der Routine ist in diesem Fall unnötig - er sorgt dafür, daß das Ergebnis von *StrFunc* auch dann 4 Zeichen Länge nicht übersteigt, wenn mit *Insert*, *Copy* usw. gearbeitet wurde.

```
STRFUNC1:@
2DC6 BF6002        MOV     DI,0260             ; Adresse der Variablen
2DC9 1E            PUSH    DS                    S auf den Stack
2DCA 57            PUSH    DI
2DCB 83EC05        SUB     SP,+05              ; 5 Bytes Platz fürs Ergebnis
2DCE E8CEFF        CALL    STRFUNC             ; Aufruf
2DD1 B104          MOV     CL,04               ; Maximal 4 Zeichen kopieren
2DD3 E825E4        CALL    11FB                ; StackToStr -> Variable S
2DD6 E90000        JMP     2DD9
2DD9 33C0          XOR     AX,AX               ; Exitcode 0
2DDB E8ABDE        CALL    0C89                ; Ende via HALT
```

Der Stack vor dem Aufruf von *StackToStr*:



StackToStr entfernt beide übergebenen Parameter wieder vom Stack*), danach hat SP also wieder den ursprünglichen Wert ($4000).

Derselbe Mechanismus wird auch bei Aufrufen innerhalb von Unterprogrammen verwendet. In der Version 3.0 hängt die Belastung des Stacks also tatsächlich vom Ergebnistyp der jeweiligen Stringfunktion ab.

Version 5.0

Abgesehen von den PUSH- und POP-Befehlen für das BP-Register und der Tatsache, daß die ersten $3C Bytes des Datensegments durch typisierte Konstanten von *System* belegt sind - die Variable *S* also ab der Adresse $003C im Speicher steht - gibt es keine Unterschiede.

Realzahlen als Funktionsergebnisse

Funktionsergebnisse des Typs *Real* werden über die Register AX/BX/DX zurückgeliefert, innerhalb der Funktion reserviert der Compiler sechs Bytes auf dem Stack für die temporäre Speicherung.

Die Version 3.0 verwendet dieselben Register, liefert Ergebnisse aber über den Stack zurück. Analog zu den anderen Funktionstypen wird der dafür nötige Platz auf dem Stack (der auch zur Zwischenspeicherung dient) von der *aufrufenden* Routine geschaffen.

Mit allen weiteren Einzelheiten beschäftigt sich das Kapitel 2 - dieser Mini-Abschnitt steht hier also eigentlich nur der Vollständigkeit halber (und für den Eintrag im Inhaltsverzeichnis).

Parameter und lokale Variablen

Offiziell unterscheidet die Sprachdefinition von Pascal nur zwischen Wert- und Variablenparametern - technisch gesehen haben wir es aber mit drei Möglichkeiten zu tun:

- Wertparameter mit einem Umfang von maximal 4 bzw. 6 Bytes (alle Integertypen inklusive *LongInt* sowie *Char*, Aufzählungstypen und *Real*) werden vor dem Aufruf in direkter Form auf den Stack geladen.

*) Weil das Funktionsergebnis hier auf dem Stack zurückgeliefert wird, gibt die *aufrufende* Routine den entsprechenden Platz wieder frei.

- Strukturierte Wertparameter (Strings, Records und Arrays) mit mehr als 4 Bytes Umfang: Hier lädt der Prozessor vor dem Aufruf einen Zeiger auf den Stack. Der Startcode der aufgerufenen Routine belegt lokal Speicherplatz (d.h. auf dem Stack) und kopiert den Parameter über eine Auswertung des Zeigers.
- Variablenparameter (alle Typen): Der Prozessor lädt einen Zeiger auf den Stack, die aufgerufene Routine greift über diesen Zeiger direkt auf die entsprechende Variable zu.

Jede Routine, die Parameter deklariert, endet unabhängig von der Art der Übergabe mit dem Befehl `RET $xx`, der die übergebenen Parameter beim Rücksprung wieder vom Stack entfernt.

Für lokale Variablen belegt die aufgerufene Routine selbständig Platz auf dem Stack, der vor dem Rücksprung durch das Zurücksetzen des Stackzeigers wieder freigegeben wird.

Einfache Wertparameter bei Prozeduren

Ein Beispiel für eine Prozedur, die einen einzigen Wertparameter des Typs *Integer* erwartet und eine einzige lokale Variable desselben Typs deklariert:

```
program WertProcDemo;
var  A: Integer;

procedure WertProc(x: Integer);
var  y: Integer;
begin
  y := x;
end;

begin
  A := 4;
  WertProc(A);
end.
```

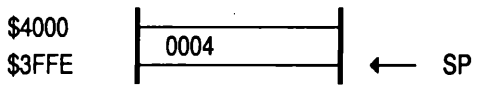
Das Ergebnis der Übersetzung durch den Compiler:

```
WERTPROCDEMO:WERTPROC:
** begin
0000 55          PUSH   BP
0001 89E5        MOV    BP,SP
0003 83EC02      SUB    SP,+02            ; 2 Bytes für Y
** y := x;
0006 8B4604      MOV    AX,[BP+04]        ; Parameter X
0009 8946FE      MOV    [BP-02],AX        ; -> Variable Y
** end;
000C 89EC        MOV    SP,BP             ; lokale Variablen vom Stack
000E 5D          POP    BP
000F C20200      RET    0002              ; Parameter X vom Stack
```

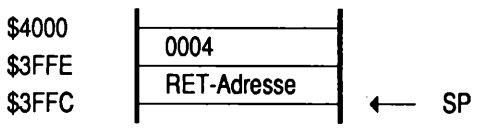
```
WERTPROCDEMO:@:
** begin
0012 9A0000653A     CALL   SYSTEM:@              ; Initialisierung
0017 89E5           MOV    BP,SP                 ; V 5.0: PUSH BP / MOV BP,SP
** A := 4;
0019 C70600000400   MOV    Word Ptr [A],0004
** WertProc(A);
001F FF360000       PUSH   [A]                   ; Wert von A auf den Stack
0023 E8DAFF         CALL   WERTPROC              ; Aufruf
** end.
0026 89EC           MOV    SP,BP                 ; V 5.0: MOV SP,BP / POP BP
0028 31C0           XOR    AX,AX                 ; Exitcode 0
002A 9AF301653A     CALL   SYSTEM:HALT           ; Programmende
```

Die Adressierung der lokalen Variablen *y* und des übergebenen Parameters *x* geschieht hier auf ähnliche Weise wie bei der Zwischenspeicherung von Funktionsergebnissen. Der Vollständigkeit halber sind die einzelnen Schritte noch einmal detailliert wiedergegeben, wobei wir wiederum von der Standardgröße des Stacks ausgehen (SP = $4000 zu Beginn des Programms):

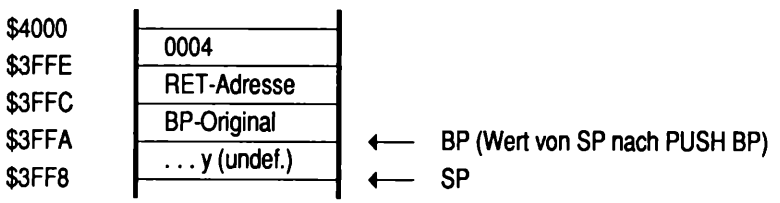
- Vor dem Aufruf von *WertProc* wird der zu übergebende Wert (d.h. die Variable A) auf den Stack gebracht.



- Der Aufruf der Prozedur bringt die Rücksprungadresse zum Hauptprogramm auf den Stack und erniedrigt SP um weitere 2 Bytes. Der Stack vor der Ausführung des ersten Befehls von *WertProc*:



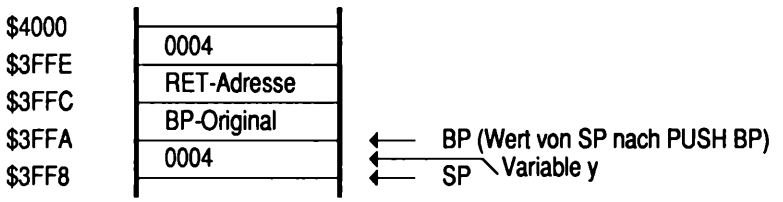
- Die ersten drei Befehle von *WertProc* bringen das BP-Register auf den Stack, weisen BP den neuen Wert von SP zu und erniedrigen SP danach um zwei Bytes, d.h. schaffen Platz für die lokale Variable *y*.



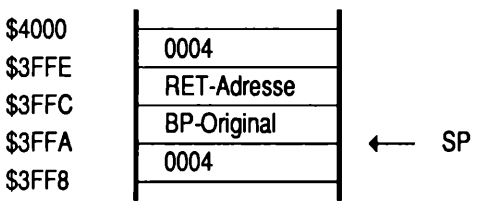
- Die nächsten beiden Befehle machen sich eine spezielle Eigenschaft des BP-Registers zunutze: Bei Indizierungen über BP stellt der 8086 implizit das Präfix SS: voran, adressiert also das Stacksegment.

  Die Routine »weiß«, daß sie als *near* (also mit 2 Bytes Rücksprungadresse) aufgerufen wurde und daß sich der Originalwert von BP ebenfalls auf dem Stack befindet. BP hat momentan den Wert $3FFA: mit `Word Ptr [BP+04]` werden folglich die Speicherzellen SS:$3FFE/F angesprochen, d.h. der übergebene Parameter *x*.

  Ähnliches gilt für die lokale Variable *y*: Sie wurde als erste (und in unserem Beispiel einzige) deklariert - der für sie reservierte Speicherbereich endet deshalb direkt unterhalb der Adresse, auf die BP zeigt. Da sie als *Integer* zwei Bytes belegt, wird sie mit `Word Ptr [BP-02]` angesprochen.



- Mit `MOV SP,BP` wird das SP-Register wieder auf den Wert $3FFA gesetzt, der folgende POP-Befehl stellt den ursprünglichen Inhalt von BP wieder her und erhöht SP auf $3FFC.



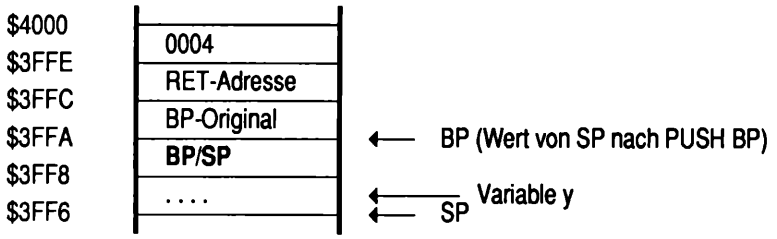
- Für den Rücksprung zum Hauptprogramm muß die Routine neben der Art ihres Aufrufs (*near* oder *far*) zusätzlich »wissen«, wieviele Bytes an Parametern auf den Stack gelegt wurden: Der Befehl `RET 0002` liest die Rücksprungadresse vom Stack (wobei SP um 2 auf $3FFE erhöht wird) und erhöht SP *danach* noch einmal um zwei. SP hat nun wieder denselben Wert wie zu Beginn des Hauptprogramms ($4000).

Der Zugriff auf *x* und *y* in diesem Beispiel folgt einem Prinzip, das für alle lokalen Variablen und Parameter gilt und auch in ähnlicher Form im Handbuch steht: *»Parameter werden über [BP+$xx] adressiert, lokale Variablen mit [BP-$xx]«*. Wieso die Anführungszeichen? Weil diese Aussage nur bedingt stimmt. Oder

auch*): Bitte warten Sie mit eigenen Experimenten, bis Sie den Abschnitt »Vorsicht mit inline!« gelesen haben.

Version 3.0

Die Adressierung des Parameters *x* verläuft hier genauso: Zwischen dem übergebenen Wert und der Adresse, auf die BP innerhalb der Routine zeigt, liegen die RET-Adresse und der Originalwert von BP auf dem Stack (4 Bytes). Da die Version 3.0 aber den neu gesetzten Wert von BP zusätzlich noch auf den Stack bringt (wieso eigentlich?), wird die lokale Variable *y* hier nicht mit `[BP-02]`, sondern mit `[BP-04]` adressiert:



Allgemein: In den Versionen 4.0 und 5.0 beginnt der Speicherplatz für lokale Variablen direkt unterhalb der Adresse, auf die BP zeigt; in der Version 3.0 dagegen 2 Bytes unterhalb dieser Adresse.

Ansonsten ist der mit der Version 3.01A für die Prozedur *WertProc* erzeugte Code funktional identisch:

```
WERTPROCDEMO:WERTPROC:
2D9F 55          PUSH   BP
2DA0 8BEC        MOV    BP,SP
2DA2 55          PUSH   BP        <== das macht den Unterschied aus
2DA3 E90000      JMP    2DA6
2DA6 4C          DEC    SP                    ; 2 Bytes Platz für y
2DA7 4C          DEC    SP
2DA8 8B4604      MOV    AX,[BP+04]            ; Parameter X
2DAB 8946FC      MOV    [BP-04],AX              -> Variable y
2DAE E90000      JMP    2DB1
2DB1 8BE5        MOV    SP,BP                 ; lokale Variablen vom Stack
2DB3 5D          POP    BP
2DB4 C20200      RET    0002                  ; Parameter X vom Stack
```

Variablenparameter

Das folgende Beispiel soll zusätzlich verdeutlichen, in welcher Reihenfolge mehrere Parameter und lokale Variablen gespeichert werden:

*) ... und noch etwas dozierender, als es dieses Kapitel leider sowieso schon ist...

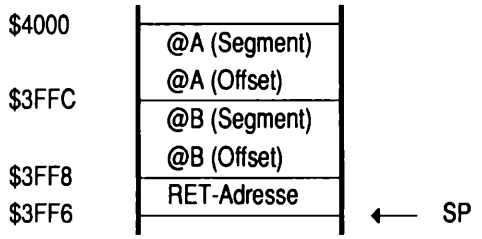
```
program VarProcDemo;
var
  A, B: Integer;

procedure Var2Proc(var I,J: Integer);
var
  x,y: Integer;
   z : Integer;
begin
  x:= 4;  y:= 6;  z:= 8;
  I := x;
  J := y;
end;

begin
  Var2Proc(A,B);
end.
```

Das Ergebnis der Übersetzung durch den Compiler:

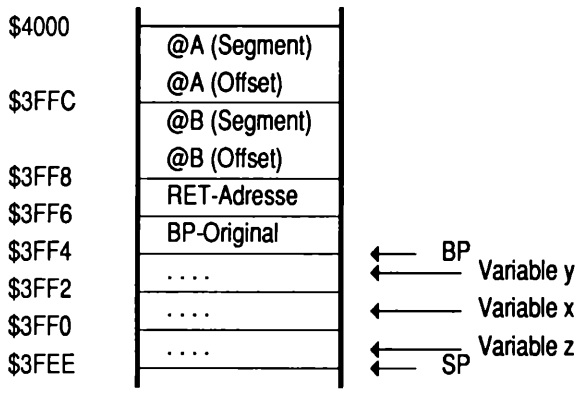
```
VARPROCDEMO:VAR2PROC:
0000 55            PUSH   BP
0001 89E5          MOV    BP,SP
0003 83EC06        SUB    SP,+06                 ; 6 Bytes lokale Variablen
0006 C746FC0400    MOV    Word Ptr [BP-04],0004 ; x := 4;
000B C746FE0600    MOV    Word Ptr [BP-02],0006 ; y := 6;
0010 C746FA0800    MOV    Word Ptr [BP-06],0008 ; z := 8;
** I:= x;
0015 8B46FC        MOV    AX,[BP-04]             ; Wert von x -> AX
0018 C47E08        LES    DI,[BP+08]             ; Adresse von A -> ES:DI
001B 268905        MOV    ES:[DI],AX             ; -> x -> A
** J:= y;
001E 8B46FE        MOV    AX,[BP-02]             ; Wert von y -> AX
0021 C47E04        LES    DI,[BP+04]             ; Adresse von B -> ES:DI
0024 268905        MOV    ES:[DI],AX             ; -> y -> B
** end;
0027 89EC          MOV    SP,BP                  ; lokale Variablen vom Stack
0029 5D            POP    BP
002A C20800        RET    0008                   ; Parameter vom Stack

VARPROCDEMO:@:
002D 9A0000673A    CALL   SYSTEM:@               ; Initialisierung
0032 89E5          MOV    BP,SP                  ; V 5.0: PUSH BP / MOV BP,SP
** Var2Proc(A,B);
0034 BF0000        MOV    DI,0000                ; Adresse von A
0037 1E            PUSH   DS                       auf den Stack (DS:0000)
0038 57            PUSH   DI
0039 BF0200        MOV    DI,0002                ; Adresse von B
003C 1E            PUSH   DS                       auf den Stack (DS:0002)
```

```
003D 57          PUSH  DI
003E E8BFFF      CALL  VAR2PROC          ; Aufruf
** end.
0041 89EC        MOV   SP,BP             ; V 5.0: MOV SP,BP / POP BP
0043 31C0        XOR   AX,AX             ; Exitcode 0
0045 9AF301673A  CALL  SYSTEM:HALT       ; Programmende
```

Das aufrufende Programm bringt hier zwei Zeiger auf die Variablen *A* und *B* in der Reihenfolge ihrer Angabe auf den Stack, der vor der Ausführung des ersten Befehls von *Var2Proc* den folgenden Inhalt hat:



Nach der Belegung der lokalen Variablen *x*, *y* und *z*, die der Compiler offensichtlich so anordnet, wie es ihm gerade in den Kram paßt*), sieht der Stack so aus:



Eine kurze Beschreibung des Ablaufs: BP hat innerhalb der Routine den Wert $3FF4. Nach dem Setzen der Werte für *x*, *y* und *z* lädt der Befehl `LES DI,[BP+08]` das Register DI mit dem Inhalt der Speicherzellen SS:$3FFC/D und das Register ES mit SS:$3FFE/F - der Offset-Anteil des übergebenen Zeigers auf die Variable *A* *muß* also auf der niedrigeren Adresse stehen, damit die Sache funktioniert. Nach dem Setzen von *A* wird derselbe Ablauf mit dem zweiten Zeiger (d.h. der Adresse von *B*) wiederholt. Der Befehl `MOV SP,BP` erhöht SP wieder auf den Wert $3FF4 und

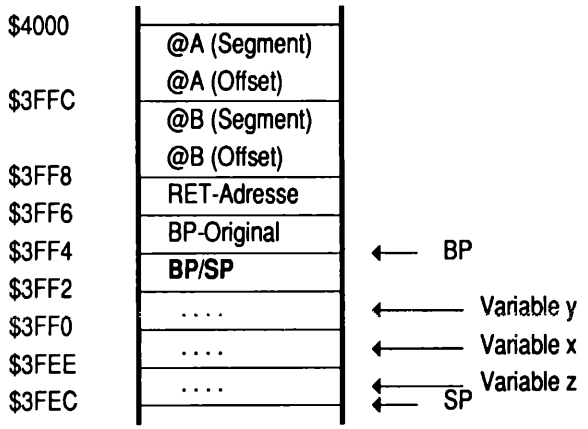
*) Etwas technischer: Bei mehreren Deklarationen mit einem Typ von rechts nach links, ansonsten in der Reihenfolge der Deklarationen - eine absolute Katastrophe für direkte Zugriffe durch Maschinenprogramme.

entfernt so die Variablen *x*, *y* und *z* vom Stack, `POP BP` stellt den ursprünglichen Inhalt von BP wieder her (und erhöht SP auf $3FF6).

Wir haben hier zwei Zeiger (mit jeweils 4 Bytes) übergeben: Der Befehl `RET 0008` liest die Rücksprungadresse (erhöht SP damit auf $3FF8) und addiert danach 8 zum momentanen Wert von SP hinzu. Als Ergebnis hat SP wieder den Wert $4000, d.h. die beiden übergebenen Parameter sind entfernt.

Version 3.0

Abgesehen davon, daß *Var2Proc* auch in diesem Fall das BP-Register ein zweites Mal auf den Stack bringt und sich die relativen Adressen lokaler Variablen damit um zwei nach unten verschieben, ergibt sich ein grundsätzlicher Unterschied: Lokale Variablen werden ordentlich in der Reihenfolge ihrer Deklaration adressiert - **unabhängig davon, ob man sie nacheinander oder zusammen deklariert.** Nach dem Setzen der Variablen *x*, *y* und *z* hat der Stack hier also den folgenden Inhalt:



Ansonsten arbeitet die Version 3.0 nach denselben Prinzipien: Das Hauptprogramm bringt die Zeiger in der Reihenfolge der Deklaration auf den Stack; die Behandlung dieser Parameter (und die Offsets zu BP) innerhalb von *Var2Proc* sind gleich:

```
VARPROCDEMO:VAR2PROC:
2D9F 55          PUSH   BP
2DA0 8BEC        MOV    BP,SP
2DA2 55          PUSH   BP
2DA3 E90000      JMP    2DA6

** begin
2DA6 83EC06      SUB    SP,+06          ; 6 Bytes lokale Variablen
2DA9 B80400      MOV    AX,0004
2DAC 8946FC      MOV    [BP-04],AX      ; x := 4;
2DAF B80600      MOV    AX,0006
```

```
2DB2 8946FA        MOV    [BP-06],AX          ; y := 6;
2DB5 B80800        MOV    AX,0008
2DB8 8946F8        MOV    [BP-08],AX          ; z := 8;

** I := x;
2DBB C47E08        LES    DI,[BP+08]          ; Adresse von A -> ES:DI
2DBE 8B46FC        MOV    AX,[BP-04]          ; Wert von x -> AX
2DC1 268905        MOV    ES:[DI],AX          ; x -> A

** J := y;
2DC4 C47E04        LES    DI,[BP+04]          ; Adresse von B -> ES:DI
2DC7 8B46FA        MOV    AX,[BP-06]          ; Wert von y -> AX
2DCA 268905        MOV    ES:[DI],AX          ; y -> B

** end;
2DCD E90000        JMP    2DD0
2DD0 8BE5          MOV    SP,BP               ; Lokales vom Stack
2DD2 5D            POP    BP
2DD3 C20800        RET    0008                ; Parameter vom Stack
```

Strukturierte Wertparameter

Daß hier schon wieder ein String zur Demonstration verwendet wird, hat seinen Grund*): Im Gegensatz zu »temporären« Speicherbereichen (siehe »Stringfunktionen«, Seite 56) reserviert der Compiler hier tatsächlich nur soviel Platz, wie für einen zuvor definierten Stringtyp benötigt wird.

```
program WertStrucDemo;
type
  Str80 = string[80];
var
  s: Str80;

procedure WertStruc(WStr: Str80);
var
  Ch: Char;
begin
  Ch := WStr[3];
end;

begin
  WertStruc(s);    { s ist undefiniert - spielt hier aber keine Rolle }
end.
```

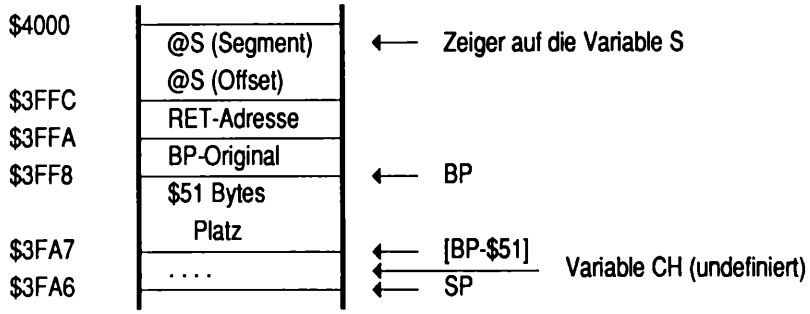
*) (Ganz ehrlich - ich habe auch schon mal ein Programm geschrieben, das mit Records arbeitet.)

Das Ergebnis der Übersetzung durch den Compiler:

```
WERTSTRUCDEMO:WERTSTRUC:
** begin
0000 55          PUSH   BP
0001 89E5        MOV    BP,SP
0003 83EC52      SUB    SP,+52            ; 82 Bytes lokale Variablen
0006 C47E04      LES    DI,[BP+04]        ; Adresse von S -> ES:DI
0009 06          PUSH   ES                ; auf den Stack
000A 57          PUSH   DI
000B 8D7EAF      LEA    DI,[BP-51]        ; Zieladresse -> DI
000E 16          PUSH   SS                ; zusammen mit SS
000F 57          PUSH   DI                  auf den Stack
0010 B85000      MOV    AX,0050           ; maximal 80 Zeichen
0013 50          PUSH   AX
0014 9A8702663A  CALL   SYSTEM:STRNCOPY   ; Kopie von S -> WSTR

** Ch := WStr[3];
0019 8A46B2      MOV    AL,[BP-4E]        ; = BP-81 + 3
001C 8846AE      MOV    [BP-52],AL        ; -> Variable CH

** end;
001F 89EC        MOV    SP,BP             ; WSTR und CH vom Stack
0021 5D          POP    BP
0022 C20400      RET    0004              ; Zeiger auf S vom Stack

WERTSTRUCDEMO:@:
** begin
0025 9A0000663A  CALL   SYSTEM:@
002A 89E5        MOV    BP,SP             ; V 5.0: PUSH BP / MOV BP,SP
** WertStruc(s);
002C BF0000      MOV    DI,0000           ; Adresse von S auf den Stack
002F 1E          PUSH   DS                  (V 4.0: DS:0000)
0030 57          PUSH   DI                  (V 5.0: DS:003C)
0031 E8CCFF      CALL   WERTSTRUC         ; Aufruf
** end.
0034 89EC        MOV    SP,BP             ; V 5.0: MOV SP, BP / POP BP
0036 31C0        XOR    AX,AX             ; Exitcode 0
0038 9AF301663A  CALL   SYSTEM:HALT       ; Programmende
```

Das Hauptprogramm arbeitet hier auf dieselbe Weise wie bei der Übergabe eines var-Parameters: Vor dem Aufruf der Prozedur *WertStruc* wird ein Zeiger auf die Variable *s* auf den Stack gelegt.

Innerhalb von *WertStruc* brauchen wir 82 Bytes Platz für lokale Variablen (ein Längenbyte und 80 Zeichen für den String, 1 Byte für die Variable *Ch*). Der Stack nach der Erniedrigung von SP:



Der Speicherplatz für die »Lokalkopie« des übergebenen Strings wird also *vor* allen explizit deklarierten lokalen Variablen reserviert.

Die Routine *StrNCopy* (»String Number Copy«, eine Anleihe von mir an die Programmiersprache »C«) erwartet drei Parameter: eine Quelladresse, eine Zieladresse und die Maximalzahl zu kopierender Zeichen. Für die Ermittlung der Zieladresse wird der Befehl LEA (»Load Effective Address« = Lade die sich ergebende Adresse) verwendet, dessen Syntax eine der vielen kleinen Ungereimtheiten des 8086 darstellt: `LEA DI, [BP-$51]` lädt das Register DI mit dem Wert von BP ($3FF8) minus $51 (= $3FA7) - den Inhalt der Speicherzelle SS:$3FA7, die eigentlich über [BP-$51] adressiert wird, ignoriert der Prozessor dabei komplett. Als Segment-Anteil der Zieladresse wird das Register SS auf den Stack gebracht: *StrNCopy* kopiert also den String *s* in den Bereich SS:$3FA7..3FF7. (Wobei maximal die ersten 80 Zeichen von *s* bearbeitet werden - wenn der String länger als 80 Zeichen ist, schneidet ihn *StrNCopy* nach dem 80. Zeichen ab; ist er kürzer, bleibt der Rest des reservierten Speicherplatzes undefiniert).

Erst nach dieser Kopie kommt der Prozessor zu den eigentlichen Anweisungen von *WertStruc*. Wir haben den Index in *WStr* über eine Konstante angegeben (`WStr[3]`) - da der Compiler weiß, daß *WStr* mit der Adresse `[BP-$51]` beginnt, hat er das Ergebnis der Adressberechnung (`[BP-$51+03]` = `[BP-$4E]`) bereits eingesetzt.

Für Indizierungen der Form *WStr(x)* erzeugt der Compiler übrigens den kompaktesten Code, den ich jemals bei einer Hochsprache gesehen habe, nämlich:

```
MOV  DI,[X]          ; Wert von X -> DI
MOV  AL,[BP+DI-$51]  ; x. Zeichen von WStr -> AL
```

Zurück zum Ablauf von *WertStruc*: Nach der Speicherung des Zeichens in *Ch* wird SP wieder auf den Wert $3FF8 gesetzt (Entfernung von *WStr* und *Ch*), `POP BP` stellt den ursprünglichen Inhalt von BP wieder her (und erhöht SP auf $3FFA). Der Befehl `RET 0004` erhöht SP erst um zwei auf $3FFC und wirft danach den übergebenen Zeiger vom Stack, d.h. erhöht SP wieder auf den Wert $4000.

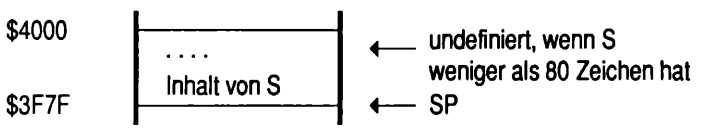
Version 3.0

Die Version 3.0 macht bei Wertübergaben keinen Unterschied zwischen einfachen Werten mit 2 oder 4 Bytes und Strings, die aufrufende Routine legt den Parameter

in jedem Fall (und in voller Größe) auf den Stack. Dementsprechend gewaltig fallen hier die Unterschiede aus - einen speziellen Startcode für »Lokalkopien« gibt es nicht:

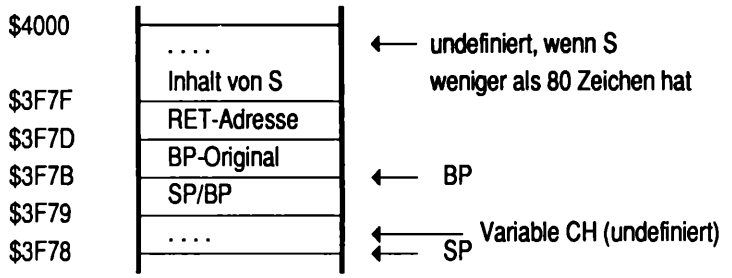
```
WERTSTRUCDEMO:WERTSTRUC:
** begin
2D9F 55          PUSH   BP
2DA0 8BEC        MOV    BP,SP
2DA2 55          PUSH   BP
2DA3 E90000      JMP    2DA6
2DA6 4C          DEC    SP                  ; 1 Byte Platz für CH
** Ch := WStr[3];
2DA7 8A4607      MOV    AL,[BP+07]          ; WStr[3] -> AL
2DAA 32E4        XOR    AH,AH               ; naja...
2DAC 8846FD      MOV    [BP-03],AL          ; -> CH
** end;
2DAF E90000      JMP    2DB2
2DB2 8BE5        MOV    SP,BP               ; CH vom Stack
2DB4 5D          POP    BP
2DB5 C25100      RET    0051                ; S vom Stack (81 Bytes)

WERTSTRUCDEMO:@
** begin
2DB8 BF6002      MOV    DI,0260             ; Adresse von S auf
2DBB 1E          PUSH   DS                    den Stack (DS:0260)
2DBC E80DE4      CALL   11CC                ; StrToStack
2DBF B150        MOV    CL,50               ; 80 Bytes muß er lang sein
2DC1 E879E4      CALL   123D                ; SetStrParamSize
2DC4 E8D8FF      CALL   WERTSTRUC           ; Aufruf
** end.
2DC7 E90000      JMP    2DCA
2DCA 33C0        XOR    AX,AX               ; Exitcode 0
2DCC E8BADE      CALL   0C89                ; Programmende via HALT
```

Der Aufruf von *StrToStack* im des Hauptprogramm lädt nur den tatsächlichen Inhalt des Strings *s* auf den Stack (und erniedrigt SP entsprechend). Da die momentane Länge von *s* dem Compiler nicht bekannt ist und *WertStruc* diesen String über den Befehl RET $xx wieder vom Stack entfernen soll, wird über *SetStrParamSize* nachgeholfen: Diese Routine sorgt über ein Umkopieren von *s* dafür, daß der Stack vor dem Aufruf von *WertStruc* unabhängig von der aktuellen Stringlänge so aussieht:



(Hätte man das mit einer eigenen Routine, die SP gleich (und unabhängig von der Stringlänge) erniedrigt, nicht etwas einfacher haben können? Hätte man. Weil *SetStrParamSize* nur für diese »Nachkorrektur« verwendet wird und damit überflüssig würde, ließen sich sogar einige Bytes in der Laufzeitbibliothek einsparen).

Genug gebesserwissert. Nach der Ausführung der ersten drei Befehle von *WertStruc* hat der Stack den folgenden Inhalt:



Auch hier hat der Compiler die Adresse des dritten Zeichens von *WStr* bereits bei der Übersetzung berechnet: Zwischen der Adresse, auf die BP momentan zeigt, und dem Anfang des Strings liegen 4 Bytes (BP-Original und RET-Adresse), auf `WStr[3]` wird also mit `[BP+07]` zugegriffen.

Da *WertStruc* für den übergebenen String nicht selbst Speicherplatz reserviert, verläuft der Zugriff auf die Variable *Ch* genauso wie sonst auch: *Ch* ist die einzige lokale Variable der Prozedur und wird mit `[BP-03]` adressiert.

Der Befehl `MOV SP,BP` setzt SP wieder auf den Wert $3F7B und entfernt *Ch* (zusammen mit der Kopie von BP/SP) vom Stack, die Befehle `POP BP` und `RET $51` erhöhen SP auf $3F7D bzw. $3F7F und schließlich wieder auf $4000.

Allgemein läßt sich der Unterschied zwischen den beiden Versionen des Compilers hier wohl am besten so formulieren: Der Zugriff auf das Element *x* eines strukturierten Wertparameters erfolgt ab der Version 4.0 über *[BP-Parameterlänge+x]*, in der Version 3.0 dagegen über *[BP+x]*.

Unterschiede zwischen Funktionen und Prozeduren

Wie im Abschnitt »Parameterlose Funktionen« (Seite 50) erläutert, reserviert eine Funktion automatisch lokalen Speicherplatz für die Zwischenspeicherung ihres Ergebnisses - und zwar ähnlich wie bei »Lokalkopien« *vor* allem anderen. Mit dem Ergebnis, daß sich die relativen Adressen aller lokalen Variablen *um die Größe des Funktionsergebnisses verschieben.*

Das folgende Programm soll der Prozedur *WertProc* (siehe »Parameter und lokale Variablen«) so weit wie möglich ähneln, um einen direkten Vergleich zu ermöglichen:

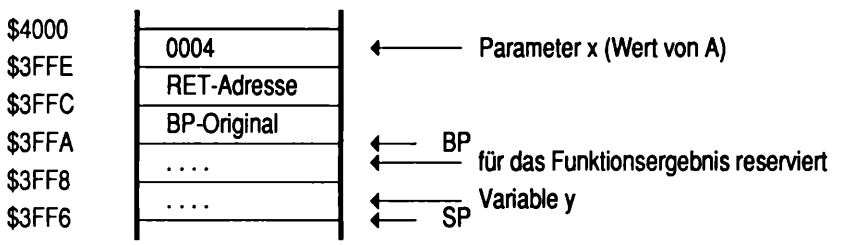
```
program WertFuncDemo;
var
  A: Integer;
```

```
function WertFunc(x: Integer): Integer;
var
  y: Integer;
begin
  y := x;
end;          { Funktionsergebnis ist undefiniert ... }

begin
  A := 4;
  A:= WertFunc(A);
end.
```

Das Ergebnis der Übersetzung durch den Compiler:

```
WERTFUNCDEMO:WERTFUNC:
** begin
0000 55          PUSH   BP
0001 89E5        MOV    BP,SP
0003 83EC04      SUB    SP,+04              ; 2 Bytes y, 2 fürs Ergebnis
** y := x;
0006 8B4604      MOV    AX,[BP+04]          ; Parameter x -> AX
0009 8946FC      MOV    [BP-04],AX          ; -> Variable y
** end;
000C 8B46FE      MOV    AX,[BP-02]          ; Funktionsergebnis -> AX
000F 89EC        MOV    SP,BP               ; lokale Variablen vom Stack
0011 5D          POP    BP
0012 C20200      RET    0002                ; Parameter x vom Stack
```

Das (hier nicht gelistete) Hauptprogramm bringt den Wert der Variablen *A* vor dem Aufruf von *WertFunc* auf den Stack, der nach den ersten drei Befehlen von *WertFunc* so aufgeteilt ist:



Die Variable *y* wird also innerhalb von *WertFunc* nicht mit `[BP-02]`, sondern mit `[BP-04]` adressiert.

Dasselbe gilt für Funktionen mit anderen einfachen Ergebnistypen. Vor allen lokalen Variablen wird der Speicherplatz für das Funktionsergebnis reserviert:

- Ergebnistypen *Byte*, *Char*, *ShortInt* oder Unterbereiche mit Ordinalzahlen zwischen 0 und 255: die Adressierung der ersten lokalen Variablen verschiebt sich um eins;

- Ergebnistypen *Word* und *Integer*: Verschiebung um zwei;
- Ergebnistypen *LongInt* und *Pointer*: Verschiebung um vier;
- Ergebnistyp *Real*: Verschiebung um sechs.

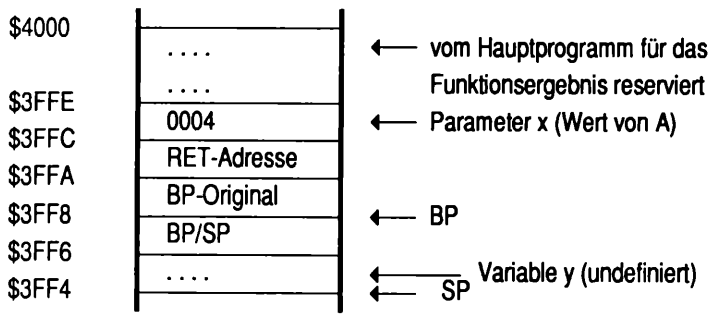
Funktionen mit strukturierten Ergebnistypen (Strings, Arrays und Records) bekommen dagegen einen Zeiger auf das »Zielgebiet« als *Parameter* übergeben und kommen deshalb ohne eine Zwischenspeicherung des Ergebnisses auf dem Stack aus - hier verschiebt sich die Adressierung lokaler Variablen nicht. (Zugriffe auf Parameter bleiben ebenfalls dieselben wie bei Prozeduren, weil die »Zieladresse« als allererste auf den Stack gebracht wird und damit »am weitesten von BP entfernt« liegt).

Version 3.0

In der Version 3.0 reserviert die *aufrufende* Routine den Platz zur Zwischenspeicherung des Funktionsergebnisses auf dem Stack, das damit »oberhalb von BP« liegt - die Adressierung lokaler Variablen ist hier also für Funktionen dieselbe wie bei Prozeduren:

```
WERTFUNCDEMO:WERTFUNC:
** begin
2D9F 55          PUSH   BP
2DA0 8BEC        MOV    BP,SP
2DA2 55          PUSH   BP
2DA3 E90000      JMP    2DA6
2DA6 4C          DEC    SP                ; 2 Bytes Platz für y
2DA7 4C          DEC    SP
** y := x;
2DA8 8B4604      MOV    AX,[BP+04]        ; Parameter x -> AX
2DAB 8946FC      MOV    [BP-04],AX        ; -> Variable y
2DAE E90000      JMP    2DB1
** end;
2DB1 8B4606      MOV    AX,[BP+06]        ; Funktionsergebnis -> AX
2DB4 8BE5        MOV    SP,BP             ; Y und BP/SP vom Stack
2DB6 5D          POP    BP
2DB7 C20400      RET    0004              ; 2 Bytes x, 2 fürs Ergebnis
```

Nach den ersten drei Befehlen von *WertFunc* hat der Stack den folgenden Inhalt:



Der Compiler behandelt hier das Funktionsergebnis sozusagen als zusätzlichen Parameter, der zur Zwischenspeicherung »mißbraucht« und am Ende der Routine zusammen mit den anderen Parametern wieder vom Stack entfernt wird*).

Vorsicht mit **inline**!

Diese Überschrift hätte auch heißen können: »Wer Parameter und lokale Variablen über **inline** direkt adressiert, ist selber schuld«. Das folgende Programm demonstriert fast alle Widrigkeiten, die sich bei der Adressierung lokaler Variablen über das BP-Register ergeben können:

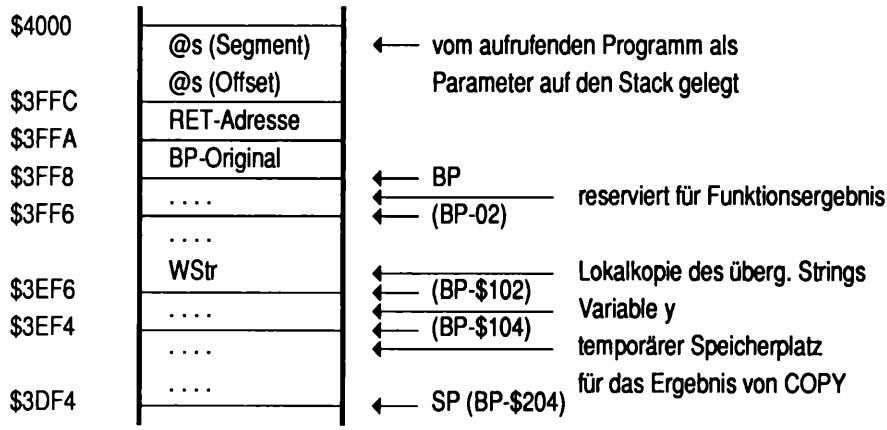
```
program WertStrFuncDemo;
var
  A: Integer;
  s: string;

function WertStrFunc(WStr:string): Integer;
{ Liefert das Ergebnis 4, wenn WStr mehr als 3 Zeichen enthält,
  ansonsten die Länge von WStr }
var
  y: Integer;
begin
  Wstr := Copy(WStr,1,4);
  y := Length(WStr);    { damit wir auch die lokale Variable drin haben }
  WertStrFunc := y;
end;

begin
  s := 'Ein String';
  A:= WertStrFunc(s);
end.
```

Die Funktion *WertStrFunc* besteht zwar nur aus drei Anweisungen - von denen sind aber zwei im Hinblick auf die Erzeugung maximaler Entropie ausgesucht worden (oder wie auch immer man das nennen soll):

*) Das gilt allerdings nur für Funktionen mit einfachen Ergebnistypen, die dieses Ergebnis direkt in den Registern des Prozessors zurückliefern. Ergebnisse der Typen *Real* und *String* bleiben auf dem Stack - was aber an der Adressierung lokaler Variablen nichts ändert.
Dieser Zwitter ist das Ergebnis eines weiteren Geniestreichs der 3.0-Entwickler: um sich die sonst übliche Trennung zwischen Funktionen und Prozeduren im Codegenerator (und damit einen weiteren Zweig) zu sparen, bekommen Funktionen einfach einen Pseudo-Parameter untergeschoben - mit einem ansonsten illegalen Namen und am lexikalischen Parser vorbei.

```
WERTSTRFUNCDEMO:WERTSTRFUNC:
** begin
0000 55          PUSH  BP
0001 89E5        MOV   BP,SP
0003 81EC0402    SUB   SP,0204          ; $204 Bytes Platz
0007 C47E04      LES   DI,[BP+04]       ; Adresse von s -> ES:DI
000A 06          PUSH  ES                 und als Quelladresse
000B 57          PUSH  DI                 auf den Stack
000C 8DBEFEFE    LEA   DI,[BP+FEFE]     ; SS:BP-$102
0010 16          PUSH  SS                 als Zieladresse auf
0011 57          PUSH  DI                 den Stack
0012 B8FF00      MOV   AX,00FF          ; maximal 255 Zeichen
0015 50          PUSH  AX
0016 9A87026C3A  CALL  SYSTEM:STRNCOPY  ; Kopie s -> SS:BP-$102 ...
```

Der Startcode von *WertStrFunc* reserviert $204 Bytes auf dem Stack und erzeugt wie gehabt eine »Lokalkopie« der als *WStr* übergebenen Stringvariablen. Nehmen wir die Gesamtaufteilung des Stacks vorweg, damit der Überblick bei den folgenden Operationen nicht ganz untergeht:



Offensichtlich reserviert der Compiler den Speicherplatz für das Funktionsergebnis und die »Lokalkopie« des übergebenen Strings *vor* der lokalen Variablen *y*, den »temporären« Speicherplatz für Stringfunktionen aber *danach*.

```
**  WStr := Copy(WStr,1,4);
001B 8DBEFCFD    LEA   DI,[BP+FDFC]     ; SS:BP-$204
001F 16          PUSH  SS               ; als Zieladresse auf
0020 57          PUSH  DI                 den Stack
0021 8DBEFEFE    LEA   DI,[BP+FEFE]     ; @WStr (SS:BP-$102)
0025 16          PUSH  SS                 als Quelladresse auf
0026 57          PUSH  DI                 den Stack
0027 B80100      MOV   AX,0001          ; ab dem ersten Zeichen
002A 50          PUSH  AX
002B B80400      MOV   AX,0004          ; vier Zeichen kopieren
```

```
002E 50          PUSH  AX
002F 9AB9026C3A  CALL  SYSTEM:STRCOPY      ; Ergebnis -> SS:BP-$204...
0034 8DBEFEFE    LEA   DI,[BP+FEFE]        ; SS:BP-$102
0038 16          PUSH  SS                    als *Ziel/Quelladresse*
0039 57          PUSH  DI                    auf den Stack
003A B8FF00      MOV   AX,00FF             ; maximal 255 Zeichen
003D 50          PUSH  AX
003E 9A87026C3A  CALL  SYSTEM:STRNCOPY     ; -> WStr (SS:BP-$102...)
```

Dieser Befehlsblock ist das Ergebnis des unscheinbaren Aufrufs von *Copy*. Hier haben die Programmierer von Borland offensichtlich etwas getrickst: Die Bibliotheksroutine *StrCopy* erwartet vier Parameter (Quelladresse, Zieladresse, Start und Länge), wobei die Zieladresse *immer auf einen temporären Speicherbereich zeigen muß* - sie bleibt nämlich gleich als Quelladresse für den folgenden Aufruf von *StrNCopy* auf dem Stack.

```
** y := Length(WStr);
0043 8A86FEFE    MOV   AL,[BP+FEFE]        ; WStr[0] (BP-$102)
0047 30E4        XOR   AH,AH
0049 8986FCFE    MOV   [BP+FEFC],AX        ; -> Variable y (BP-$104)
```

Der Aufruf von *Length* - ursprünglich nur eingefügt, damit auch die Variable *y* verwendet wird - zeigt einen echten Fortschritt gegenüber der Version 3.0. Hier wird einfach das Längenbyte ausgelesen: *Length* ist seit der Version 4.0 also eine Art Spezialfunktion, die (wie *Inc* und *Dec*) ohne Parameterübergaben und Unterprogrammaufrufe auskommt.

```
** WertStrFunc := y;
004D 8B86FCFE    MOV   AX,[BP+FEFC]        ; Variable y -> AX
0051 8946FE      MOV   [BP-02],AX            -> Funktionsergebnis
**   end;
0054 8B46FE      MOV   AX,[BP-02]          ; Funktionsergebnis -> AX
0057 89EC        MOV   SP,BP               ; alles Lokale runter
0059 5D          POP   BP                    vom Stack
005A C20400      RET   0004                ; Parameter vom Stack
```

Der Befehl `MOV SP,BP` hat hier recht gewaltige Auswirkungen: er entfernt über 500 Bytes vom Stack. Nach dem folgenden `POP BP` bleibt neben der Rücksprungadresse nur noch der einzige Parameter der Routine übrig - der Zeiger auf die Variable *s*.

Die Adressierung lokaler Variablen relativ zum Wert von BP hängt insgesamt von drei Faktoren ab:

- dem Platzbedarf des Funktionsergebnisses (soweit vorhanden);
- dem Gesamtumfang eventueller »Lokalkopien« strukturierter Wertparameter (deren Anzahl im Prinzip beliebig ist);

- und schließlich der Art, wie aufeinanderfolgende Variablen gleichen Typs deklariert sind (zusammen als *A,B: Integer;* oder getrennt als *A:Integer; B:Integer;* - vgl. Abschnitt »Variablenparameter«).

In Bezug auf numerische Funktionsergebnisse sollte man dazu noch ausgesprochen vorsichtig sein: Wer eine Funktion mit dem Ergebnistyp *Real* (Platzbedarf: 6 Bytes) deklariert und hinterher wie im Referenzhandbuch vorgeschlagen den Typ *Real* als *Extended* deklariert (Platzbedarf: 10 Bytes), schreibt plötzlich inmitten der falschen lokalen Variablen (oder gleich im Funktionsergebnis) herum. So harmlos die Ergebnisse in den meisten Fällen auch sein werden - bis man hinter diese Art von Fehlern kommt, kann schon einige Zeit vergehen.

inline mit symbolischen Namen - eine zweischneidige Lösung

Ein weiteres Zitat aus dem (mittlerweile schon arg strapazierten) Kapitel 25 des Referenzhandbuchs: »Lokale Variablen (die innerhalb einer Prozedur oder Funktion deklariert sind) werden im Stacksegment gespeichert. Hier berechnet **inline** nicht eine »absolute« Adresse, sondern erzeugt automatisch einen Relativwert zum Inhalt des BP-Registers, wodurch auch automatisch der Stack adressiert wird«.

Peinlich, aber wahr: Der letzte Halbsatz ist Unsinn und findet sich deshalb auch nur in der 4.0-Ausgabe*). Was der Compiler in diesem Fall erzeugt, ist ein (korrekter) »Offset« zum momentanen Stand des BP-Registers, der zur Konstruktion eines Zugriffs auf den Stack über BP verwendet werden *kann*.

MOV-Befehle zwischen Registern und einer über `[BP+/-$xx]` adressierten Speicherzelle werden beispielsweise so für den Prozessor codiert:

```
Zielregister/          $46        Offset (1 Byte = -128..+127)
Art der Operation      $86        Offset (2 Bytes = -32768..+32767)
   1. Byte        |  2. Byte   |    3./4. Byte
```

Der Wert des ersten Bytes legt fest, welches Register an der Aktion beteiligt ist und in welche Richtung der Transfer gehen soll. An den Bits 7 und 6 des zweiten Bytes erkennt der Prozessor, ob die folgende Relativadresse 8 oder 16 Bit umfaßt. (Andere Befehle als MOV, die über [BP+/-$xx] arbeiten, sind ähnlich aufgebaut: `LEA DI,[BP+$xx]` ist als `8D7Exx` für Offsets mit 8 Bit und als `8DBExxxx` für Offsets mit 16 Bit codiert - auch hier wird die Unterscheidung zwischen den beiden Offset-Größen über die Bits 7 und 6 des zweiten Befehlsbytes getroffen).

Wenn Sie sich nun fragen, was dieser gedankliche Ausflug soll (normalerweise kümmert man sich nicht einmal bei der Assembler-Programmierung um derartige Details): **Der Compiler nimmt keine Analyse der vorangehenden inline-Bytefolge vor und erzeugt immer Offsets mit 16 Bit.** Wer also eine kurze Routine mit DEBUG ausprobiert - etwa

*) Darin erschöpfen sich die Korrekturen aber auch.

```
-a
0100 xor ax,ax
0102 mov [bp-2],ax
0105
-u 100,104
0100 31C0          XOR     AX,AX
0102 8946FE        MOV     [BP-02],AX
-q
```

und die von DEBUG erzeugten Bytes hinterher vertrauensvoll in eine **inline**-Anweisung einsetzt

```
procedure LocalZero;
var x: Integer;
begin
  inline($31/$C0/     { XOR  AX,AX }
         $89/$46/x);  { MOV  [x],AX  ; x auf [BP-02] }
end;
```

- der sollte froh sein, wenn er sich das Ergebnis **vor** dem Ausprobieren des Programms noch einmal ansieht. Hier ist es:

```
PROGRAM:LOCALZERO:
** begin
0000 55            PUSH   BP
0001 89E5          MOV    BP,SP
0003 83EC02        SUB    SP,+02
                                     inline($31/$C0/     { XOR  AX,AX }
0006 31C0          XOR    AX,AX
                                            $89/$46/x);  { MOV [x],AX }
0008 8946FE        MOV    [BP-02],AX
000B FF89EC5D      DEC    Word Ptr [BX+DI+5DEC]
000F C3            RET
```

Eigentlich sollte hier die Befehlsfolge `MOV SP,BP` / `POP BP` stehen. Das tut sie auch - aber erst hinter dem zweiten Byte des Offsets von *x*, das der Compiler unerwünscht erzeugt hat. Augenscheinlich wird diese Extraleistung von DEBUG (und vom Prozessor) gründlich mißverstanden.

(Wo der Prozessor bei der Ausführung des RET-Befehls landet? Auf der Adresse 0000 innerhalb des momentanen Codesegments - diesen Wert haben wir der Variablen x gegeben, die sich direkt oberhalb von SP auf dem Stack befindet).

Dasselbe gilt für die Adressierung von Parametern. Auch dazu ein Beispiel:

```
DEBUG
-a
0100 les di,[bp+4]
0103 es:
0104 mov al,[di]
```

```
-u 100,105
0100 C47E04        LES     DI,[BP+04]
0103 26            ES:
0104 8A05          MOV     AL,[DI]
```

Die ermittelten Befehlsbytes in eine Prozedur eingesetzt:

```
program GetParms;
var Cx: Char;

procedure x (var  Ch: Char);
begin
  inline($C4/$7E/Ch/     { LES DI,CH [BP] ; Adresse der Variablen }
         $26/$8A/$05);   { MOV AL,ES:[DI] ; Wert von Ch -> AL }
end;

begin
  x(Cx);
end.
```

Und das haarsträubende Ergebnis:

```
GETPARMS:X:
** begin
0000 55            PUSH    BP
0001 89E5          MOV     BP,SP
                                        inline($C4/$7E/Ch/     { LES DI,CH [BP] }
0003 C47E04        LES     DI,[BP+04]
                                                $26/$8A/$05);  { MOV AL,ES:[DI] }
0006 00268A05      ADD     [058A],AH    <== überzähliges Byte
** end;
000A 89EC          MOV     SP,BP
000C 5D            POP     BP
000D C20400        RET     0004
```

Es gibt zwei Möglichkeiten zur Lösung dieses Problems, die leider beide nicht der Weisheit letzter Schluß sind:

- Verwenden Sie bei der Ermittlung von Befehlscodes mit DEBUG grundsätzlich »Dummy«-BP-Offsets, die größer als $80 sind (z.B. $102 anstelle von $02). DEBUG erzeugt dann Prozessorbefehle, die Offsets mit 16 Bit erwarten. Es spielt dabei keine Rolle, ob der vom Compiler hinterher eingesetzte Wert innerhalb oder außerhalb des Bereichs -128..+127 liegt (die »kurze« Codierung mit 8 Bit ist lediglich eine zusätzliche Möglichkeit zur Optimierung).

  Mit dieser Methode werden **inline**-Routinen zwar meistens um ein oder zwei Bytes länger, als sie unbedingt sein müßten - wenn Sie aber nicht gerade in einer Schleife immer wieder auf lokale Parameter zugreifen, dürften die paar Mikrosekunden zu verschmerzen sein. Außerdem sind Sie vor unangenehmen

Überraschungen sicher, wenn tatsächlich einmal ein voller 16-Bit-Offset erforderlich sein sollte.

- Für Risikofreudige: Lassen Sie die von DEBUG ermittelten Befehlscodes, wie sie sind, und machen Sie von der Möglichkeit Gebrauch, die »automatische Größenbestimmung« von **inline** über eine vorangestellte spitze Klammer außer Kraft zu setzen, d.h. begrenzen Sie den Offset explizit auf einen Wert mit 8 Bit:

```
procedure LocalZero;
var x: Integer;
begin
  inline($31/$C0/       { XOR  AX,AX }
         $89/$46/<x);   { MOV  [x],AX  ; x auf [BP-02] }
end;
```

In diesem Beispiel erzeugt der Compiler für die Angabe `<x` nur noch das Byte $FE - also einen »kurzen« Offset. Der Nachteil: Wenn die Routine durch »Lokalkopien«, Stringvariablen usw. mehr als $80 Bytes lokaler Variablen belegt, funktioniert die Sache natürlich nicht mehr - in den meisten Fällen erhalten Sie aber lediglich merkwürdige Ergebnisse (anstelle von Systemabstürzen).

Genauso gefährlich: numerische const-Deklarationen

Da der Compiler bei der Deklaration einer numerischen Konstanten über **const** immer das kleinstmögliche Speicherformat verwendet (also je nach Größe des Wertes ein Byte, ein Wort oder einen *LongInt*) und diese Werte ebenfalls rücksichtslos in **inline**-Anweisungen einsetzt, haben wir hier dasselbe Problem:

```
const
  Num1 = 1;
  Num300 = 300;

  ...
   inline ($xx/$yy/Num1/       { erzeugt ein einzelnes Byte }
           $xx/$yy/Num300/     { erzeugt ein Wort mit 2 Bytes }
    ...
```

Hier sollte man *grundsätzlich* die Operatoren < und > verwenden, um die Anzahl der vom Compiler erzeugten Bytes festzulegen und sich »Prozessorbefehle« wie `XOR AX,[BX+SI+$A4DE]` zu ersparen - speziell deshalb, weil Konstanten im Laufe einer Programmentwicklung ihre Werte manchmal recht häufig ändern. (Täten sie es nicht, könnte man sich ihre namentliche Deklaration schließlich sparen).

inline-Deklarationen

Wie in Kapitel 25 des Referenzhandbuchs beschrieben, erzeugt der Compiler für als **inline** *deklarierte* Routinen keinen Start- oder Endcode. Eventuelle Parameter

werden wie beim Aufruf einer normalen Routine auf den Stack gebracht, danach fügt der Compiler die entsprechende **inline**-Bytefolge ein.

Achtung! Bitte stellen Sie sich die nächste halbe Seite durchgehend fettgedruckt vor - die Unterschiede und Fallstricke haben es nämlich in sich:

- Der Compiler unterscheidet zwischen **inline**-Anweisungen und **inline**-Deklarationen über **begin** und **end**. Für eine Routine, die lokale Variablen deklariert und/oder mit **begin** anfängt, wird *immer* Start- und Endcode erzeugt - also auch dann, wenn ihr Anweisungsteil nur aus einem **inline** besteht.
- Der **inline**-Code wird direkt (und ohne CALL) eingesetzt, der Stack enthält also keine RET-Adresse. Dementsprechend verschiebt sich die Adressierung übergebener Parameter.
- Speicherplatz für eventuelle »Lokalkopien« und/oder Funktionsergebnisse wird nicht automatisch reserviert - dafür muß die Routine selbst sorgen.
- Da kein Endcode erzeugt wird, muß die Routine übergebene Parameter selbst vom Stack entfernen.

Und schließlich: Hinter der unscheinbaren Aussage des Referenzhandbuchs, daß sich übergebene Parameter nicht über ihre Namen adressieren lassen, verbirgt sich die Falltür des Jahres: **Die Namen formaler Parameter werden bei inline-Deklarationen vom Compiler ignoriert.** Was das bedeutet? Bei der Compilierung einer **inline**-Routine wie

```
procX (a:Integer);
inline($8B/$46/<a/        { MOV AX, a   ; via [BP+xx] }
      ....
```

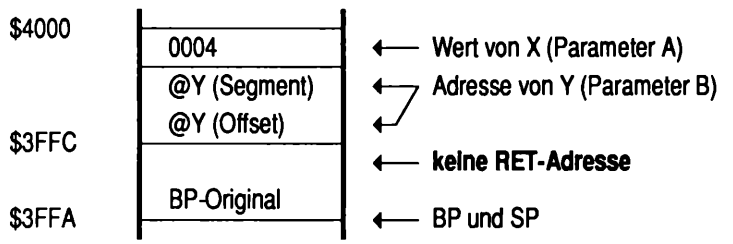
- erhalten Sie entweder die Fehlermeldung »Unbekannter Bezeichner«, weil der Compiler den formalen Parameternamen *a* ignoriert hat und deshalb keine Variable dieses Namens kennt;
- oder Sie haben eine *globale Variable* des Namens *a* deklariert, deren Adresse der Compiler hier rücksichtslos einsetzt. Falls Sie wie im vorletzten Abschnitt vorgeschlagen mit einer »Teiladressierung« (`<a`) gearbeitet haben, können Sie noch von Glück reden - der Compiler meldet sich in diesem Fall hartnäckig mit »inline Error«, weil diese Operation für *Adressen* nicht erlaubt ist.

Zurück zu erfreulicheren Aspekten. Das folgende Beispiel demonstriert zwei funktionsfähige **inline**-Deklarationen:

```
program inlineDeclDemo;
var
  X,Y,Z: Integer;
```

```
procedure proc1(a: Integer; var b: Integer);
{ Entspricht der Pascal-Anweisung b := a; }
inline(  $55/            { PUSH    BP                              }
         $89/$E5/        { MOV     BP,SP                           }
         $C4/$7E/$02/    { LES     DI,[BP+02]  ; Adresse b -> ES:DI }
         $8B/$46/$06/    { MOV     AX,[BP+06]  ; Wert a -> AX      }
         $26/$89/$05/    { MOV     ES:[DI],AX  ; Zuweisung         }
         $5D/            { POP     BP                              }
         $83/$C4/$06);   { ADD     SP,+06      ; Parameter vom Stack }

function func1(a:Integer; var b:Integer): Integer;
{ Entspricht der Pascal-Anweisung func1 := a + b; }
inline( $5F/             { POP     DI          ; Offset von b -> DI  }
        $07/             { POP     ES          ; Segment von b -> ES }
        $58/             { POP     AX          ; Wert a -> AX        }
        $26/$03/$05);    { ADD     AX,ES:[DI]  ; Addition            }

begin
  X:= 4; proc1(X,Y);
  Z:= func1(X,Y);
end.
```

Die Routine *proc1* verwendet dieselbe Art der Adressierung wie eine Pascal-Prozedur: Nach der Speicherung von BP werden die Parameter über dieses Register gelesen. Der Stack innerhalb von *proc1*:



Die Funktion *func1* macht es sich da wesentlich einfacher: Weil hier ebenfalls keine RET-Adresse auf dem Stack liegt, können wir die Parameter einfach mit POP herunterholen. Außerdem spart sich die Routine eine Zwischenspeicherung des Funktionsergebnisses (und eine entsprechende Reservierung von Platz auf dem Stack) - sie liefert ihr Ergebnis direkt über das AX-Register zurück.

Das Ergebnis der Übersetzung durch den Compiler:

```
INLINEDECLDEMO:@:
** begin
0000 9A00006B3A     CALL   SYSTEM:@              ; Initialisierung
0005 89E5           MOV    BP,SP
0007 C70600000400   MOV    Word Ptr [X],0004     ; X := 4;
```

```
** proc1(X,Y);
000D FF360000      PUSH   [X]                ; vom Compiler erzeugt:
0011 BF0200        MOV    DI,0002              Wert von X und Adresse
0014 1E            PUSH   DS                   von Y auf den Stack
0015 57            PUSH   DI

0016 55            PUSH   BP                 ; Kopie der inline-
0017 89E5          MOV    BP,SP                Prozedur (unverändert)
0019 C47E02        LES    DI,[BP+02]
001C 8B4606        MOV    AX,[BP+06]
001F 268905        MOV    ES:[DI],AX
0022 5D            POP    BP
0023 83C406        ADD    SP,+06

** Z:= func1(X,Y);
0026 FF360000      PUSH   [X]                ; vom Compiler erzeugt:
002A BF0200        MOV    DI,0002              Wert von X und Adresse
002D 1E            PUSH   DS                   von Y auf den Stack
002E 57            PUSH   DI

002F 5F            POP    DI                 ; Kopie der inline-Funktion
0030 07            POP    ES                   (unverändert)
0031 58            POP    AX
0032 260305        ADD    AX,ES:[DI]

0035 A30400        MOV    [Z],AX             ; vom Compiler: Zuweisung

** end.
0038 89EC          MOV    SP,BP
003A 31C0          XOR    AX,AX              ; Exit-Code 0
003C 9AF3016B3A    CALL   SYSTEM:HALT        ; Programmende
```

Prozeduren und Funktionen in Maschinensprache

Fangen wir mit einer etwas nationalbewußteren*) Version der Standardfunktion *UpCase* an - das Original läßt wie üblich die deutschen Umlaute unberücksichtigt:

```
; UPCASE.ASM - Ersatz der Funktion UpCase durch eine
; Version, die auch deutsche Umlaute berücksichtigt

CODE    Segment Public Byte
        PUBLIC UpCase
        ASSUME cs:CODE
```

*) Jawoll!

```
; -------------------------------
; function Upcase(Ch: Char): Char;
; -------------------------------
UpCase  proc near
        pop bx              ; Rücksprungadresse -> BX
        pop ax              ; Ch -> AX
        cmp al,'a'
        jb UpSet            ; -> kleiner als 'a'
        cmp al,'z'
        jbe SetUp           ; in 'a'..'z': Umsetzen
        cmp al,'ä'
        jnz Up1
        mov al,'Ä'
Up1:    cmp al,'ö'          ; hier wird einfach
        jnz Up2             ; weiter verglichen,
        mov al,'Ö'          ; um die zusätzlichen
Up2:    cmp al,'ü'          ; Sprungbefehle ein-
        jnz UpSet           ; zusparen. Nicht gerade
        mov al,'Ü'          ; edel, aber wirksam.
        jmp UpSet
SetUp:  sub al,'a'-'A'      ; "Übersetzung"
UpSet:  jmp bx              ; "RET"
UpCase  endp
CODE    ends
        end
```

So merkwürdig die Behandlung der Rücksprungadresse und des übergebenen Parameters *Ch* nach den vorangegangenen Besprechungen auch aussehen mag - wir sparen uns damit die Zwischenspeicherung von BP und die spätere Korrektur des Stacks. (In der Laufzeitbibliothek der Version 3.0 wird dieses Verfahren recht häufig verwendet).

Wenn Sie diese Routine als *far* aufrufen wollen, wird es etwas komplizierter, weil der 8086 einen Befehl wie `jmp far <Registerpaar>` nicht kennt. Die einfachste Lösung sieht ungefähr so aus:

```
        pop bx              ; Offset-Anteil der Rücksprungadresse
        pop es              ; Segment-Anteil
        pop ....            ; Parameter vom Stack
        ....
        push es
        push bx             ; Rücksprungadresse wieder drauf
         ....
         ....               ; Ausführung der Routine
        ret
```

Nach der Übersetzung mit MASM (oder einem anderen Assembler) läßt sich die neue Version von *UpCase* folgendermaßen in einem Pascal-Programm verwenden:

```
program UpcaseDemo1;

function UpCase(Ch: Char): Char; external;
{$L UPCASE}     { '.OBJ' wird vom Linker selbständig angehängt }

var
  x: Integer;
  s: string;
begin
  s:= 'Funktioniert auch für äöü.';
  for x:= 1 to Length(s) do
   s[x] := UpCase(s[x]);
  Writeln(s);
end.
```

Es gibt noch einen zweiten Grund, warum man in manchen Fällen auf »standardmäßige« Behandlung des Stacks verzichten sollte: Die von MASM für UPCASE.ASM erzeugten Befehlscodes lassen sich ohne weitere Veränderung in einer **inline**-Version dieser Funktion verwenden, wenn man den ersten und den letzten Befehl (`POP BX` bzw. `JMP BX`) wegläßt:

```
program inlineUpCase;
function UpCase(Ch:Char): Char;
inline(
       $58/          {        pop ax      ; Ch -> AX              }
       $3C/$61/      {        cmp al,'a'                          }
       $72/$1B/      {        jb UpSet    ; -> kleiner als 'a'    }
       $3C/$7A/      {        cmp al,'z'                          }
       $76/$15/      {        jbe SetUp   ; in 'a'..'z': Umsetzen }
       $3C/$84/      {        cmp al,'ä'                          }
       $75/$02/      {        jnz Up1                             }
       $B0/$8E/      {        mov al,'Ä'                          }
       $3C/$94/      { Up1:   cmp al,'ö'  ; hier wird einfach     }
       $75/$02/      {        jnz Up2     ; weiter verglichen,    }
       $B0/$99/      {        mov al,'Ö'  ; um die zusätzlichen   }
       $3C/$81/      { Up2:   cmp al,'ü'  ; Sprungbefehle ein-    }
       $75/$07/      {        jnz UpSet   ; zusparen. Nicht gerade }
       $B0/$9A/      {        mov al,'Ü'  ; edel, aber wirksam.   }
       $EB/03/90/    {        jmp UpSet   ; [ + NOP ]             }
       $2C/$20);     { SetUp: sub al,'a'-'A' ; "Übersetzung"      }
                     { UpSet: equ *                               }

begin
  Writeln(UpCase('ä'));
end.
```

Was dieses Beispiel leider auch demonstriert: Selbst eine einfache Funktion wie *UpCase* ist für eine **inline**-Routine eigentlich schon zu lang. Dafür sind die

Geschwindigkeitsunterschiede allerdings erstaunlich: die Assembler-Version UPCASE.ASM braucht ziemlich exakt doppelt solange wie ihr **inline**-Gegenstück.

Eine deutsche Version von *UpperCase*

Die in Kapitel 25 des Referenzhandbuchs vorgestellte Routine zur Umsetzung eines Strings in Großbuchstaben hat dasselbe Manko wie die Originalversion von *UpCase* - weil die deutschen Umlaute außerhalb des Bereichs 'a'..'z' liegen, bleiben sie unberücksichtigt. Das folgende Programm ist eine einfache Erweiterung des Originals:

```
; UPPER.ASM - Übersetzung eines Strings in Groß-/Kleinbuchstaben
; unter Berücksichtigung der deutschen Umlaute

CODE     Segment Public Byte
         PUBLIC Upper
         ASSUME CS:Code
; --------------------------------
; function Upper(s:string): string;
; --------------------------------
Upper    Proc near          ; *** im Gegensatz zum Original: NEAR ***
         push bp
         mov bp,sp
         push ds
         lds si,[bp+04]     ; Adresse von s -> DS:SI
         les di,[bp+08]     ; Zieladresse -> ES:DI
         cld
         lodsb              ; Längenbyte von s
         stosb              ; Ziel-Längenbyte
         mov cl,al          ; das ist das Längenbyte
         xor ch,ch
         jcxz UpDone        ; ==> Länge 0
ULoop:   lodsb              ; ein Zeichen von s -> AL
         cmp al,'a'         ; kleiner als 'a' ?
         jb isupper         ; ==> keine Übersetzung
         cmp al,'z'         ; kleiner/gleich 'z'?
         jb islow
         cmp al,'ä'
         jnz U2
         mov al,'Ä'
         jmp isupper
U2:      cmp al,'ö'
         jnz U3
         mov al,'Ö'
         jmp isupper
U3:      cmp al,'ü'
         jnz isupper
```

```
        mov al,'Ü'
        jmp isupper
islow:  sub al,'a'- 'A'      ; "Übersetzung"
isupper: stosb               ; AL -> Zielstring
        loop ULoop           ; "until CX = 0"
UpDone: pop ds
        pop bp
        ret 4
Upper   endp
CODE    ends
        end
```

Die Deklaration von *Upper* in einem Pascal-Programm:

```
program UpperDemo;

function Upper(s:string): string; external;
{$L UPPER}   { '.OBJ wird vom Linker selbständig angehängt }

begin
  Writeln(Upper('Funktioniert auch für äöü.'));
end.
```

Weil wir hier mit mehreren Parametern arbeiten (und eine Funktion dieses Umfangs wirklich nicht mehr als **inline** codiert werden sollte), behandeln wir den Stack »standardmäßig«, d.h. adressieren die Parameter über BP.

Ein Einschub für Geschwindigkeitsfanatiker: 50000 »Übersetzungen« eines Strings mit 62 Zeichen Länge dauern auf einem AT mit 12 MHz

- 40 Sekunden mit UPCASE.ASM bzw. UPCASE.OBJ, von denen die entsprechende Schleife (`for x:= 1 to Length(XStr) do ...`) 22 Sekunden beansprucht;
- 31 Sekunden mit der **inline**-Version von *UpCase* und derselben Schleife;
- 15 Sekunden mit *Upper*.

Daß die Umsetzung eines gesamten Strings nur doppelt so schnell wie die zeichenweise Bearbeitung über eine Schleife ausgeführt wird, ist zwar erstaunlich, hat aber seinen Grund: Wie im Abschnitt »Stringfunktionen« (Seite 56) beschrieben, stellt der Compiler erst einmal einen »temporären« Speicherbereich für das Funktionsergebnis zur Verfügung - das Ergebnis von *Upper* wird bei einer Zuweisung also ein zweites Mal kopiert.

Eine Version von *Upper* als *Prozedur*, die den String als **var**-Parameter übergeben bekommt, erledigt dieselben 50000 Umwandlungen in 9 Sekunden:

```
; UPCPROC.ASM - Übersetzung eines Strings in Groß-/Kleinbuchstaben
; unter Berücksichtigung der deutschen Umlaute
; Version 2 als Prozedur mit VAR-Parameter
```

```
CODE      Segment Public Byte
          PUBLIC UpperProc
          ASSUME CS:Code
; ---------------------------------
; procedure UpperProc(var s:string);
; ---------------------------------
UpperProc Proc near
          push bp
          mov bp,sp
          push ds
          lds si,[bp+04]      ; Adresse von s -> DS:SI
          push ds
          pop es              ; Zieladresse (ES:DI)
          mov di,si           ; ist dieselbe
          cld
          lodsb               ; Längenbyte von s
       ....
       ....                   ; derselbe Code wie bei UPPER.ASM
       ....
UpDone:   pop ds
          pop bp
          ret 4
UpperProc endp
CODE      ends
          end
```

Versuchen Sie mit dieser Version von *Upper* einmal das folgende Programm:

```
program UpperProcDemo;

          { *** var-Deklaration von s absichtlich "vergessen" *** }
procedure UpperProc(s:string); external;
{$L UPCPROC}

begin
  UpperProc('Funktioniert auch für äöü.');
  Writeln  ('Funktioniert auch für äöü.');
end.
```

Das Ergebnis des Aufrufs von *Writeln* ist:

```
FUNKTIONIERT AUCH FÜR ÄÖÜ.
```

Diese wundersame Verwandlung liegt daran, daß der Compiler identische Stringkonstanten nur einmal speichert und *UpperProc* die Adresse übergeben hat, ab der die Zeichenfolge »Funktioniert auch für äöü.« im Codesegment gespeichert ist. Das bringt uns zum nächsten (und letzten) Abschnitt des Kapitels:

Pseudo-var-Parameter und die Folgen

Daß der Compiler bei jeder Wertübergabe eines strukturierten Parameters Befehle zur Erzeugung einer »Lokalkopie« voranstellt, läßt sich nicht verhindern - auch dann nicht, wenn dieser Parameter nur gelesen wird. Deklarieren wir den Parameter dagegen als **var**, dann ist die Übergabe von Konstanten oder Zwischenergebnissen verboten.

Wie die Routinen *Upper* und *UpperProc* des vorangegangenen Abschnitts demonstrieren, läßt sich diese Grenze mit Maschinenroutinen überwinden - welcher Unsinn mit übergebenen Adressen innerhalb eines Maschinenprogramms angestellt wird, entzieht sich der Kontrolle des Compilers.

Anders gesagt: *Upper* (bzw. das Original *UpperCase*) ist ein miserables Beispiel für ein Referenzhandbuch*) - wer auf der Grundlage dieser Funktion eine eigene Routine schreibt, die einen übergebenen String nicht nur liest, sondern auch verändert, wird unter Umständen ein (blaues) Wunder erleben.

Natürlich wird es Leute geben, die diese »Sicherheitslücke« unbedingt ausnutzen *müssen* - schon allein deshalb, weil es sie gibt. Nur aus diesem Grund seien hier die Möglichkeiten und Unmöglichkeiten aufgelistet:

- Solange es sich bei dem übergebenen Parameter um eine Variable oder eine typisierte Konstante handelt, ist (wie bei einer **var**-Deklaration) praktisch jede Manipulation erlaubt.

- Für Zwischenergebnisse von Stringoperationen reserviert der Compiler jeweils $100 Bytes Platz auf dem Stack, deshalb sind hier im Prinzip ebenfalls beliebige Veränderungen möglich. (Zwischenergebnisse für Operationen mit Records oder Arrays gibt es in diesem Sinne nicht).

- String*konstanten* können zeichenweise verändert werden, wenn man die entsprechenden Nebeneffekte (wie in *UpperProcDemo*) berücksichtigt (und nicht vergißt, daß der Prozessor dabei im Codesegment herumschreibt).

- **Jede Längenvergrößerung einer untypisierten Stringkonstanten führt zu einem Systemabsturz.** Wie im Abschnitt »Typisierte und untypisierte Konstanten« (Seite 43) gezeigt, speichert der Compiler bei Anweisungen wie

  ```
  StrVar := 'Hallo!';
  Writeln('Hallo!');
  ```

  die Zeichenfolge *Hallo!* direkt vor dem Code der jeweiligen Routine im Codesegment. Eine Verlängerung dieses Strings überschreibt den Anfang der Routine.

*) Bis auf einen nebulösen Hinweis hat sich daran auch in der 5.0-Ausgabe des Referenzhandbuchs nichts geändert, weil jede halbwegs schlüssige Erklärung den Mechanismus der »Lokalkopien« inklusive der Zieladressen-Übergabe, dem automatischen Reservieren lokalen Speicherplatzes durch den Compiler usw. zur Grundlage haben müßte.

Kapitel 2
Der Coprozessor 8087

Die meisten Zeitschriftenartikel, in denen der 8087 in irgendeiner Form vorkommt, beschränken sich auf den Hinweis, daß dieses Wundertier alle Fließkomma-Berechnungen übernimmt, die Ausführung um den Faktor 10 (oder mehr) beschleunigt und das Programm kürzer macht. Das Referenzhandbuch von Turbo Pascal listet noch die Speicherformate auf, gibt eine Erklärung zum »Überlauf des Rechenstacks« ab und hält das Thema dann ebenfalls für erledigt.

Was es im Prinzip auch ist. Das folgende Programm zeigt, wie verblüffend einfach die Multiplikation zweier Werte im Format *Extended* ausfällt:

```
program MultExtended;
{$N+}
const
  X: Extended = 2.0;
  Y: Extended = 4.0;
var
  Z: Extended;

begin
  Z := X * Y;
end.
```

Das Ergebnis der Übersetzung durch den Compiler:

```
MULTEXTENDED:@:
** begin
0000 9A0000633A     CALL   SYSTEM:@            ; Initialisierung
0005 9AAF02633A     CALL   SYSTEM:INIT87       ; Init 8087
000A 89E5           MOV    BP,SP
** Z := X * Y;
000C 9B             WAIT
000D DB2E0A00       FLD    TByte Ptr [Y]       ; Wert von Y laden
0011 9B             WAIT
0012 DB2E0000       FLD    TByte Ptr [X]       ; Wert von X laden
0016 9B             WAIT
0017 DEC9           FMULP  ST(1),ST            ; Multiplikation
0019 9B             WAIT
001A DB3E1400       FSTP   TByte Ptr [Z]       ; Ergebnis in Z speichern
001E 9B             WAIT
**  end.
001F 89EC           MOV    SP,BP
0021 31C0           XOR    AX,AX               ; Exitcode 0
0023 9AF301633A     CALL   SYSTEM:HALT         ; Programmende
```

Abgesehen vom Aufruf der Routine SYSTEM:INIT87 und den massenhaft eingestreuten WAIT-Befehlen hat die gesamte Operation denselben Grad an Komplexität wie das Addieren zweier Integerwerte.

Dieses Kapitel beschäftigt sich deshalb nicht mit der Umrechnung von Mantissen und der Konvertierung von Formaten, sondern mit dem »Drumherum«:

- Programmiermodell des 8087 (Registerfolge, TOS, Statusbits usw.);
- Einbindung des 8087 in ein Turbo Pascal-Programm (Interrupts $02, $75 und die zusätzliche Initialisierungsroutine in *System*);
- Übergabe von Parametern und Ergebnissen;
- dem Variablentyp *Real*;
- und schließlich der direkten Programmierung des Coprozessors.

Was den letzten Punkt betrifft, gleich eine »Enttäuschung« vorweg: Auch eine Routine, die wirklich nur aus Fließkomma-Operationen[1] besteht, läßt sich mit direkter Programmierung des 8087 maximal um rund 100% beschleunigen - im Gegensatz zu vielen anderen Compilern, die mit maximal 2 Operanden für den 8087 arbeiten und deshalb selbst Additionsketten in viele kleine Stücke zerlegen müssen, nutzt Turbo Pascal sämtliche Register des 8087: je komplexer die Operation ist, desto effizienter wird der erzeugte Code.

Version 5.0

Wie bereits im Abschnitt »Typisierte und untypisierte Konstanten« des Kapitels 1 angedeutet, unterscheidet sich der letztendlich ausgeführte Code (zumindest bei einer einfachen Addition von Realzahlen) nicht sonderlich voneinander - in der zuvor gezeigten Form bekommen Sie *MultExtended* allerdings nur zu sehen, wenn Sie das Programm *vorher*

- auf einem System mit installiertem Coprozessor via DEBUG bzw. SYMDEB vollständig ausführen lassen,
- danach das DS-Register wieder auf das Datensegment setzen (für die Arbeit mit SYMDEB: `rds DATA`)
- und dann einen List-Befehl mit der entsprechenden Adresse geben (für SYMDEB: `u @`).

Zum Wie, Wo und Warum kommen wir in wenigen Seiten[2].

---

[1] Wieder einmal Apfelmännchen - mir ist für ein Kapitel über den 8087 auch nichts Besseres eingefallen.

[2] Wobei ich eigentlich versprechen sollte, daß es sich dabei um den letzten Vorwärtsverweis handelt - leider ist dem nicht so.

Zum Thema »Parallelverarbeitung«

Die einfachen Modelle aus den guten alten Zeiten (wie z.B. der Arithmetikbaustein des Prozessors Z80) lassen sich recht gut mit einer Zusatzkarte des Computers vergleichen. Hier könnte die Multiplikation zweier Realzahlen ungefähr so ablaufen:

1. Der Prozessor liest die beiden Variablenwerte aus dem Speicher und sendet sie (mit entsprechenden I/O-Befehlen) zu der Zusatzkarte. Wie und wo diese Werte in der Karte gespeichert werden, interessieren weder Prozessor noch Programmierer.
2. Der Prozessor sendet der Zusatzkarte den Befehl »Multipliziere« (was im allgemeinen über eine weitere I/O-Operation geschieht).
3. Der Prozessor fragt irgendein Statussignal der Zusatzkarte solange wiederholt ab, bis die Karte signalisiert, daß die Operation beendet ist.
4. Der Prozessor holt das Ergebnis der Operation (über entsprechende I/O-Befehle) Byte für Byte ab und schreibt es in den Hauptspeicher.

Dieser Ablauf läßt sich auch mit Prozessorbefehlen darstellen:

```
        MOV   AX, Wert1          ; erster Operand
        CALL  Op_Ausgabe         ; Ausgabe
        MOV   AX, Wert2          ; zweiter Operand
        CALL  Op_Ausgabe         ; dito
        CALL  MULT_Cmdausgabe    ; Befehlsausgabe: "Multiplizieren"
MLOOP:  CALL  Kartenabfrage      ; schon fertig?
        JNZ   MLOOP              ; ... nein
        CALL  Ergebnislesen      ; ja - Ergebnis abholen
```

Hier haben wir bereits eine echte Parallelverarbeitung: Während die Zusatzkarte mit der Multiplikation beschäftigt ist, könnte der Prozessor andere Operationen ausführen. Das setzt allerdings eine entsprechende Programmierung voraus - deshalb erschöpft sich dieses Nebeneinander meist in recht einfachen Schleifen wie

```
MLOOP:  CALL  Kartenabfrage    ; schon fertig?
        JZ    M1               ; ja!
        CALL  Tastatur         ; na gut - inzwischen können wir ja mal
        JMP   MLOOP            ; die Tastatur lesen
M1:     CALL  Ergebnislesen    ; ....
```

Das Problem, daß man in irgendeiner Form auf das Ergebnis einer Rechenoperation warten muß, bleibt letztendlich auch beim Gespann 8086/87 erhalten - ansonsten haben sich die Dinge aber radikal geändert:

Beide Prozessoren lesen *denselben* Befehlsstrom (technisch: haben eine gemeinsame Verbindung mit dem Datenbus des Systems). Jeder der beiden kennt den Befehlssatz des anderen soweit, daß er weiß, welche Befehle für wen bestimmt sind. (Wieso das notwendig ist? Der 8086 muß beispielsweise wissen, wieviele

Bytes ein 8087-Befehl umfaßt, damit er sein IP-Register entsprechend erhöhen kann). Das »Senden eines Befehls zur Zusatzkarte« wird hier sozusagen im Direktverfahren erledigt.

Der 8087 kann selbständig *Daten* des Hauptspeichers lesen und neue Werte hineinschreiben (technisch: er hat einen eigenen und vollständigen Satz von Adressleitungen). Damit sich die beiden Prozessoren nicht gegenseitig ins Gehege kommen, sind sie mit einigen Direktleitungen untereinander verbunden: Der 8087 kann dem 8086 signalisieren, daß er momentan auf den Hauptspeicher zugreifen will.

Was er dagegen nicht hat, ist ein eigener Befehlszähler: für das Lesen von *Befehlen* ist er also auf den 8086 angewiesen. Einerseits kann man nur sagen: Gott sei Dank - sonst hätten wir es mit zwei völlig unabhängigen Prozessoren und dem ganzen Sack damit verbundener Probleme (Synchronisation, Kommunikation usw.) zu tun. Andererseits haben sich die Entwickler damit ein neues Problem eingehandelt: Es muß auf irgendeine Weise dafür gesorgt werden, daß *der 8086 keinen neuen 8087-Befehl liest, während der Coprozessor beschäftigt ist.* Eine weitere Direktleitung zwischen beiden Prozessoren, deren Zustand mit dem 8086-Befehl WAIT abgefragt werden kann, schlägt deshalb zwei Fliegen mit einer Klappe: über sie signalisiert der 8087 nicht nur »Operation beendet, Ergebnis kann weiterverarbeitet werden«, sondern auch (mit entgegengesetztem Pegel): »bin beschäftigt und kann keine weiteren Befehle entgegennehmen«. Was darauf hinausläuft, daß bei einer Befehlsfolge wie

```
WAIT
<Coprozessor-Befehl>
WAIT
```

der erste WAIT-Befehl den 8086 solange anhält, bis der 8087 seine Bereitschaft signalisiert hat; der zweite WAIT-Befehl hält den 8086 solange an, bis der Coprozessor meldet: »Operation beendet«. Die Notwendigkeit des ersten WAIT-Befehls im folgenden Ausriß des Beispielprogramms *MultExtended* ist damit offensichtlich: Der 8087 braucht für die Multiplikation einen Moment und kann während dieser Zeit keine weiteren Befehle entgegennehmen.

```
FMULP   ST(1),ST       ; Multiplikation
WAIT
FSTP    TByte Ptr[Z]   ; Ergebnis in Z speichern
WAIT
MOV     SP,BP          ; Stack zurück
XOR     AX,AX
CALL    SYSTEM:HALT
```

Wozu aber das Warten auf »Operation beendet« nach dem Befehl zur Speicherung? Weil sich die Entwickler der Firma Intel eine weitere Maßnahme zur Steigerung der Geschwindigkeit ausgedacht haben: *Zugriffe auf den Hauptspeicher finden versetzt statt.* Wenn der WAIT-Befehl vor `MOV BP,SP` nicht wäre, dann könnte der zeitliche Ablauf für das Schreiben der zehn Bytes von *Z* ungefähr so aussehen:

```
        8086                        8087
  --- gemeinsames Lesen des Befehls FSTP ---
     wartet                 schreibt 1. Byte von Z
  liest MOV SP,BP           holt 2. Byte für Z intern
  interne Ausführung        schreibt 2. Byte von Z
     wartet                 holt 3. Byte für Z intern
     wartet                 schreibt drittes Byte von Z
  liest XOR AX,AX           holt 4. Byte für Z intern
 interne Ausführung         schreibt 4. Byte für Z
  liest <CALL FAR>          holt 5. Byte für Z intern
    wartet                  schreibt 5. Byte für Z
    ....                       .....
```

Und nun die traurige Erkenntnis: Weil der Compiler (prinzipbedingt durch die Struktur der Sprache Pascal) nicht wissen kann und will, welche Operation nach einem Befehl wie FSTP als nächstes kommt, schließt er jeden 8087-Befehl grundsätzlich mit einem WAIT ab - schließlich könnte es sich beim nächsten 8086-Befehl um einen direkten Zugriff auf eine Adresse handeln, die der 8087 noch gar nicht gesetzt hat.

Die Register des 8087

Der 8087 verfügt über acht Rechenregister (R0..R7) auf dem Chip, die in Form eines Hardware-Stacks organisiert sind. Jedes dieser Register hat eine Breite von 80 Bit (= 10 Bytes) und kann einen Wert im Format *Single, Double, Comp* oder *Extended* speichern. Dazu kommen noch drei weitere Register*), die größtenteils bitweise strukturiert und jeweils 16 Bit breit sind. Eines dieser drei Register reserviert 3 Bits für den Stackzeiger, der in derselben Weise wie beim 8086 *vor* einer Ladeoperation erniedrigt wird. Was beim Stack des 8086 normalerweise eine Katastrophe bedeutet, ist beim 8087 der Regelfall: Wenn der Stackzeiger den Wert 0 nach unten hin unterschreitet, fängt er »oben« wieder an, d.h. mit dem Wert 7.

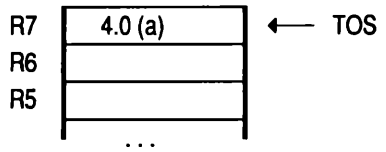
Das jeweils »oberste« Register des 8087 wird als TOS (»Top of Stack«) bezeichnet und ist an jeder Rechenoperation beteiligt, die zwei Registerinhalte miteinander verknüpft. Nach einer Initialisierung des 8087 mit dem Befehl FINIT zeigt der Stackzeiger auf das Register R0 - eine darauffolgende Ladeoperation setzt also das Register R7(!) auf den entsprechenden Wert.

Mit diesen Informationen können wir das Beispielprogramm *MultExtended* (Seite 93) schrittweise verfolgen, wobei die diversen WAIT-Befehle der Übersichtlichkeit halber nicht mehr aufgeführt sind.

*) Eigentlich sind es vier, weil man dem 8087 über ein *Environment* auch noch mitteilen kann, welchen Sprung des 8086 er bei Fließkomma-Fehlern erzwingen soll. Da dieses Register von Turbo Pascal nur gelesen wird und ansonsten hauptsächlich im »protected mode« zur Anwendung kommt, wird es in der folgenden Besprechung schlichtweg unterschlagen.

```
000D DB2E0A00        FLD    TByte Ptr [Y]          ; Wert von Y laden
```

Die Initialisierung des 8087 ist über den Aufruf der *System*-Routine INIT87 am Anfang des Hauptprogramms geschehen. Der Befehl FLD (»Float LoaD«) erniedrigt den Stackzeiger *zuerst* um eins (von 0 auf 7) und speichert den Wert von *Y* danach in dem Register, auf das der Stackzeiger jetzt zeigt:

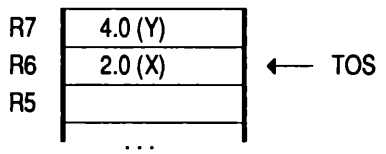


(Der einzige Unterschied zur Organisation des 8086-Stacks ist sozusagen grafischer Natur: Weil wir es hier nur mit acht Speicherzellen (= Registern) zu tun haben, die nicht Teil des Hauptspeichers sind, können wir uns die Darstellung »verkehrtherum« leisten, die eigentlich anschaulicher ist als das übliche »Wachsen des Stacks nach unten«).

Der zweite FLD-Befehl hat denselben Ablauf: Zuerst wird der Stackzeiger um eins (von 7 auf 6) erniedrigt, dann wird der Wert von *X* in das entsprechende Register geladen.

```
0012 DB2E0000        FLD    TByte Ptr [X]          ; Wert von X laden
```

Der 8087-Stack nach dieser Operation:



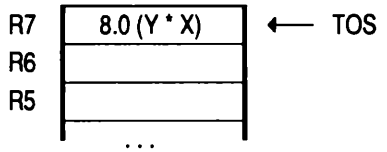
Der nächste Befehl (FMULP = »Float MULtiply Pop«) besteht aus zwei aufeinanderfolgenden Operationen:

- Multiplikation des Registers*), das sich eins *über* der momentanen Spitze des Stacks befindet (ST(1) = R7), mit dem TOS (R6) und Speicherung des Ergebnisses in ST(1).
- Erhöhung des Stackzeigers um eins (»Pop«). Der Inhalt von R6 gilt danach als »undefiniert« und ist (im Gegensatz zum 8086-Stack) auch tatsächlich nicht mehr verfügbar.

```
0017 DEC9            FMULP  ST(1),ST               ; Multiplikation
```

*) Die Syntax dieses Befehls ist dieselbe wie beim 8086, bei dem `ADD AX,BX` für »Addiere AX und BX, speichere das Ergebnis in AX« steht - das Zielregister einer binären Operation wird also zuerst angegeben.

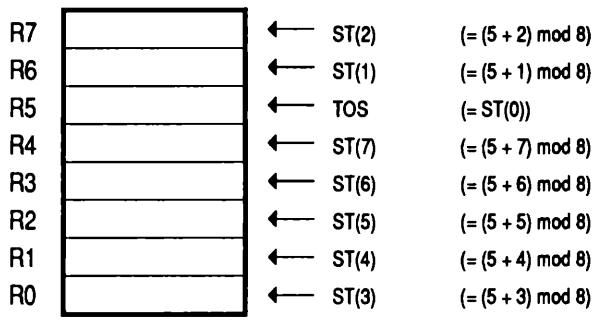
Der 8087-Stack nach der Ausführung des Befehls:



Der letzte Befehl (»Float STore Pop«) führt ebenfalls zwei Operationen nacheinander aus: Zuerst wird der Inhalt des TOS (R7) in den Hauptspeicher geschrieben, danach erhöht der 8087 den Stackzeiger um eins (von 7 auf 0) und betrachtet auch R7 nun als »undefiniert«, d.h. der Rechenstack ist wieder leer.

```
001A DB3E1400        FSTP   TByte Ptr [Z]          ; Ergebnis in Z speichern
```

Adressierungen von Registern arbeiten immer relativ zum TOS und werden Modulo 8 behandelt. Wenn der Stackzeiger beispielsweise auf R5 zeigt, dann werden die restlichen Register folgendermaßen angesprochen:



So merkwürdig und ungewohnt diese relative Adressierung erscheinen mag - für den Compiler, der bei der Analyse zusammengesetzter Ausdrücke intern sowieso mit umgekehrter polnischer Notation*) arbeitet, ist sie ideal. Ein Ausdruck wie

```
Z := A * B + C * D
```

wird bei dieser Rechenform so umgeschrieben:

```
Z := A B *  C D *  +
```

Diese Folge steht für: Multiplikation von A mit B, Zwischenspeicherung des Ergebnisses; Multiplikation von C mit D und Addieren des vorherigen Ergebnisses. Tatsächlich läßt sich dieser Ablauf direkt in 8087-Befehle umsetzen:

*) Auch als Postfix-Notation bekannt, allen Forth-Programmierern bestens vertraut und dem Rest der Welt ein Greuel. Die in letzter Zeit populär gewordene Seitenbeschreibungssprache PostScript (von Adobe Systems) verwendet dasselbe Prinzip - daher auch der Name.

```
FLD TByte Ptr [A]        ; Wert von A -> R7        R7 = TOS
FLD TByte Ptr [B]        ; Wert von B -> R6        R6 = TOS
FMULP ST(1),ST           ; R6 * R7 -> R7 / POP     R7 = TOS
```

A und B auf den Stack, Multiplikation mit FMULP: das Ergebnis bleibt im TOS (R7) stehen.

```
FLD TByte Ptr [C]        ; Wert von C -> R6        R6 = TOS
FLD TByte Ptr [D]        ; Wert von D -> R5        R5 = TOS
FMULP ST(1),ST           ; R5 * R6 -> R6 / POP     R6 = TOS
```

C und D auf den Stack, Multiplikation mit FMULP: das Ergebnis bleibt im TOS (R6) stehen.

```
FADDP ST(1),ST           ; R6 + R7 -> R7 / POP     R7 = TOS
FSTP TByte Ptr [Z]       ; TOS -> Z / POP          R0 = TOS
```

Addition der beiden obersten Register mit FADDP: übrig bleibt das Gesamtergebnis in R7, das schließlich zusammen mit einem weiteren »Pop« in der Zielvariablen Z gespeichert wird.

Dieses Beispiel demonstriert auch noch einen weiteren Vorteil des Stack-Konzepts (zumindest für den Compiler): Sowohl die erste als auch die zweite Multiplikation geschehen mit demselben Befehl (`FMULP ST(1),ST`), obwohl sich Operanden und Ergebnis jeweils in anderen Registern befinden (einmal R6/R7, beim zweiten Mal R5/R6). Für den Compiler spielt es keine Rolle, an welcher Stelle einer zusammengesetzten Operation zwei weitere Operanden miteinander verknüpft werden: er »weiß«, daß er sie einfach auf den Stack laden und das Ergebnis über `F...P ST(1),ST` berechnen lassen kann.

Das »Tag«-Register...

müßte in der deutschen Übersetzung eigentlich Registerzustands-Register heißen: Es enthält für jedes Rechenregister des 8087 zwei Zustandsbits:

15							8	7							0
R7	R7	R6	R6	R5	R5	R4	R4	R3	R3	R2	R2	R1	R1	R0	R0

Die Anordnung dieser Bits ist vom momentanen Stand des Stackzeigers unabhängig, entspricht also der physikalischen Registerfolge des 8087. (Woraus folgt, daß man erst einmal den Stackzeiger lesen und diesen Wert dann auf die entsprechende Bitposition innerhalb des Tag-Registers umrechnen muß, um den Zustand des TOS »von außen« zu bestimmen).

Jedes der acht Rechenregister kann einen von vier Zuständen annehmen, von denen drei sehr schnell abgehandelt sind:

Tag-Bits	Bedeutung
00	Register enthält einen gültigen Wert
01	Register enthält den Wert 0.0
10	»Special«: das Register enthält einen unendlichen Wert, eine denormalisierte Zahl oder eine NaN (siehe unten)
11	Register ist leer (d.h. der Inhalt ist undefiniert). Nach einer Initialisierung des 8087 haben alle acht Register diesen Zustand, d.h. das Tag-Register enthält den Wert $FFFF.

Wenn das »Tag« eines Registers den Wert 10 (= »Special«) hat, unterscheidet der 8087 zwischen drei möglichen Inhalten:

- *Unendliche Werte* werden bei der Überschreitung des Rechenbereichs oder des jeweiligen Speicherformats erzeugt. Eine Unendlichkeit hat ein Vorzeichen und wird vom 8087 auch als solche behandelt: *-1/0* ergibt *-Inf* (d.h. eine negative Unendlichkeit), *-Inf+10000* ergibt nach wie vor *-Inf* usw. Das »Tag« wird hier auf jeden Fall gebraucht, um den Registerinhalt als Unendlichkeit zu kennzeichnen - im Speicher ablegen kann man derartige Werte also nicht.

- *Denormalisierte Werte* entstehen, wenn das Ergebnis einer Rechenoperation so nahe bei 0.0 liegt, daß es im verwendeten Speicherformat nur noch mit führenden Nullen in der Mantisse ausgedrückt werden kann. Normalerweise ist eine Zahl immer so umgerechnet, daß ihre Mantisse mit einem gesetzten Bit beginnt. Die kleinste *normalisierte* Zahl, die sich im Format *Single* darstellen läßt, hat den Wert $2.0 * 2^{-126}$ (= $2.3... * 10^{-38}$). Sie hat die Bitfolge

```
(1.)000000000....          00000001
Mantisse (23 Bit)       Exponent (8 Bit)
```

wobei die führende »1« und der binäre Dezimalpunkt bei Berechnungen implizit vorangestellt werden (d.h. nicht Teil der gespeicherten Bits sind). Wenn man von dieser Festlegung abgeht (die für alle »normalisierten« Zahlen gilt), kann man auch noch kleinere Werte erfassen - im Format *Single* minimal:

```
0.00000000000000000000001  00000000
```

Hier ist nur noch die 23. Binärstelle hinter dem Komma gesetzt, die Mantisse hat damit den Wert $1 / 2^{23}$. Der Exponent hat den Wert 0 und wird als 2^{-126} gewertet, er kennzeichnet die Zahl als denormalisiert. Das Ergebnis der Berechnung $1 / 2^{23} * 2^{-126}$ ergibt ziemlich genau die im Handbuch angegebene Minimalgröße von Werten im Format *Single*, nämlich $1.4*10^{-45}$.

Weil sich für große Zahlenwerte nicht einfach binäre Dezimalstellen verschieben lassen, bleibt der größte im Format *Single* darstellbare Wert bei

```
1.11111111111111111111111  11111111
```

(also 1 + 0.5 + 0.25 + 0.125.... * 2^{126} = $3.4*10^{38}$) - aus diesem Grund sind die Rechenbereiche der Typen *Single..Extended* asymmetrisch zum Nullpunkt*). Da die Kennzeichnung des Registerinhalts als denormalisiert über den Wert des Exponenten erfolgt, ist das »Tag« im Prinzip nicht notwendig.

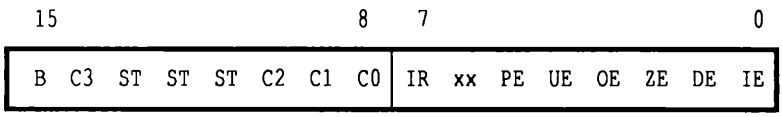
- Das Kürzel *NaN* steht für »Not a Number« - derartige »Werte« werden durch mathematisch undefinierte Rechenoperationen wie negative Wurzeln oder negative Logarithmen erzeugt. Das Zielregister einer derartigen Operation Register erhält das Tag »Special«, einen ansonsten reservierten Wert im Exponenten und einen der folgenden (standardisierten) Werte in der Mantisse:

NaN-Wert	Bedeutung
001	ungültige Quadratwurzel, z.B. *Sqrt(-1)*
002	ungültige Addition oder Subtraktion, z.B. *Inf - Inf*
004	ungültige Division, z.B. 0 / 0
008	ungültige Multiplikation, z.B. 0 * *Inf*
020	Ergebnis der Konvertierung einer *NaN* in ein anderes Format
021	Versuch, eine *NaN* mit dem Code 000 zu erzeugen
033	ungültiges Argument für eine trigonometrische Funktion
034	ungültiges Argument für eine inverse trigonometrische Funktion
036	ungültiges Argument für *Log*
037	ungültiges Argument für x^i oder x^y

Wie bei denormalisierten Zahlen ist das Tag hier eigentlich unnötig - der spezielle Wert des Exponenten reicht aus, um Verwechslungen mit »richtigen« Zahlenwerten zu vermeiden.

Das Statuswort-Register...

enthält hauptsächlich drei Elemente: den Stackzeiger (3 Bit, Werte von 0..7), verschiedene Flags (Carry, Zero, Parity), die durch Vergleiche von Fließkommazahlen gesetzt werden, und schließlich sechs »Exception Flags«, die Ausnahmezustände wie Divisionen durch Null signalisieren. Mit dem Befehl FSTSW [<Adresse>] kann es im Hauptspeicher abgelegt und dort vom 8086 gelesen werden. Die Flags für Ausnahmezustände (PE..IE) lassen sich mit dem Befehl FCLEX (»CLear EXceptions«) direkt löschen, andere direkte Zugriffe auf das Statuswort sind nicht definiert.



*) Eine vernünftige Abhandlung ist das nicht gerade. Da sich der 8087 aber um diese Bit-Hexereien automatisch kümmert, lassen wir es trotzdem damit bewenden.

B — »BUSY« - wenn dieses Bit gesetzt ist, führt der 8087 momentan einen Befehl aus. Wie man an dieses Bit herankommen soll, ohne dabei einen 8087-Befehl (nämlich FSTCW) auszuführen, ist mir ehrlich gesagt unklar*).

C3..C0 — »CONDCODE3..0« - diese Bits werden unter anderem durch Vergleiche von Registerinhalten gesetzt. Sie sind so angeordnet, daß sie mit der Befehlsfolge FSTSW <Adresse> / MOV AX, <Adresse> / SAHF direkt in die Flags des 8086 übertragen werden können. Für die Kombination 8086/87 ist dieser Weg der einzige zur Übergabe eines Vergleichsergebnisses; die Ausführungen ab 80286/287 können das Statuswort auch direkt in das Register AX einlesen.

ST — »STACKTOP« - diese drei Bits werden als zusammenhängender Wert interpretiert (MSB auf Bit 13 des Statuswortes, LSB auf Bit 11).

IR — »INTERRUPT« - wenn dieses Bit gesetzt ist, will der 8087 den 8086 wegen eines Ausnahmezustands unterbrechen (bzw. hat ihn gerade unterbrochen - außer in einer INT 02-Behandlungsroutine bekommt man dieses Bit nur dann zu sehen, wenn man alle Ausnahme-Interrupts (via IEM im Kontrollwort) unterdrückt, eine entsprechende Coprozessor-Operation und dann das Statuswort abfragt).

PE — »PRECISION EXCEPTION« - gesetzt, wenn das Ergebnis der letzten Operation auf- oder abgerundet werden mußte (z.B. bei einer Division).

UE — »UNDERFLOW EXCEPTION« - das Ergebnis der letzten Operation ist so klein (d.h. liegt so nahe bei 0.0), daß es im denormalisierten Format oder direkt als 0.0 gespeichert werden mußte.

OE — »OVERFLOW EXCEPTION« - das Ergebnis der letzten Operation ist so groß, daß es als *Inf* gespeichert werden mußte.

ZE — »ZERO EXCEPTION« - die letzte Operation war eine Division durch Null.

DE — »DENORMALIZED EXCEPTION« - die letzte Operation hat versucht, einen denormalisierten Registerinhalt als Operanden zu verwenden.

IE — »INVALID EXCEPTION« - die letzte Operation war aus irgendeinem Grund ungültig (negative Wurzel, NaN, Register leer usw).

*) Dem Datenblatt von Intel fällt dazu auch nichts Erleuchtendes ein.

Das Kontrollwort-Register...
enthält zwei Gruppen von Flags: Zum einen werden hier Genauigkeit und Rundungsrichtung festgelegt, zum anderen die Masken für Ausnahmezustände - die Flags IEM..IM bestimmen, ob der Coprozessor den 8086 bei einer »Ausnahme« mit einem Interrupt unterbricht oder ob er seine Arbeit fortsetzt. Mit dem Befehl FSTCW kann das momentane Kontrollwort im Hauptspeicher abgelegt werden, der Befehl FLDCW setzt es mit neuen Werten. Weil der 8087 keine Befehle wie »setze RC = xx« kennt, besteht eine Veränderung des Kontrollwort-Registers aus drei Schritten: FSTCW [Adresse] / Veränderung der Bits durch den 8086 / FLDCW [Adresse].

15							8	7							0
xx	xx	xx	IC	RC	RC	PC	PC	IEM	xx	PM	UM	OM	ZM	DM	IM

IC — »INFINITY CONTROL« - legt fest, welche Art von »Unendlichkeitsmathematik« verwendet wird. (Nur für mathematisch Interessierte: das projektive Modell (IC = 0) behandelt den Zahlenstrahl als geschlossenen Kreis und Unendlichkeiten deshalb ohne Vorzeichen; das affine Modell (IC = 1) betrachtet ihn dagegen als (theoretische) Gerade, es unterscheidet zwischen *-Inf* und *+Inf*).

RC — »ROUNDING CONTROL« - legt eine der vier möglichen Rundungsrichtungen fest (00 = Rundung zum nächsten geradzahligen Wert; 01 = Rundung in positive Richtung; 10 = in negative Richtung; 11 = Rundung gegen 0 mit darauffolgendem Abschneiden von Nachkommastellen)

PC — »PRECISION CONTROL« - legt fest, mit welcher Genauigkeit Rechenoperationen ausgeführt werden (11 = *Extended*, 10 = *Double*, 00 = *Single*). Der Wert 01 ist meines Wissens »reserviert«, setzt aber ebenfalls die Genauigkeit *Single*.

IEM — »INTERRUPT ENABLE MASK» - legt fest, ob der 8087 den 8086 beim Auftreten eines nicht maskierten Ausnahmezustands unterbrechen kann (IEM = 0) oder nicht (IEM = 1).

PM..IM — »PRECISION MASK«..»INVALID MASK« - wenn eines dieser Bits gesetzt ist, unterbricht der 8087 das Programm beim Auftreten des entsprechenden Ausnahmezustands *nicht* (siehe nächster Abschnitt).

Die Initialisierung des 8087

Der Befehl FINIT setzt das Tag-Register auf den Wert $FFFF (»alle Rechenregister leer«), das Statuswort auf $0000 (TOS = R0, alle Ausnahmeflags zurück) und das Kontrollwort auf den Wert $037F:

15							8	7							0
xx	xx	xx	IC	RC	RC	PC	PC	IEM	xx	PM	UM	OM	ZM	DM	IM
			0	0	0	1	1	0	?	1	1	1	1	1	1

Der 8087 rundet nach dieser Initialisierung in Richtung auf den nächsten geradzahligen Wert, verwendet intern die Rechengenauigkeit von *Extended* und reagiert auf sämtliche Ausnahmezustände nur mit dem Setzen der entsprechenden Flags im Statuswort, d.h. er unterbricht den 8086 in keinem Fall.

... durch Turbo Pascal in der Version 4.0

Im Modus {$N+} beginnt ein Pascal-Hauptprogramm grundsätzlich mit zwei Aufrufen von *System*, die direkt aufeinander folgen (und unabhängig von eventuellen weiteren Units sind):

```
** begin
0000 9A0000633A     CALL   SYSTEM:@              ; Initialisierung
0005 9AAF02633A     CALL   SYSTEM:INIT87         ; Init 8087
   < Aufruf der Initialisierungsteile weiterer Units >
000A 89E5           MOV    BP,SP                 ; Beginn des Hauptprogramms
       ....
```

Die Startadresse von INIT87*) ist im Gegensatz zu der von SYSTEM:@ variabel - sie hängt hauptsächlich davon ab, welche Fließkomma-Routinen das Programm benutzt. Die Routine führt die folgenden Schritte aus:

- Prüfung, ob ein 80x87 installiert ist. Wenn nein, wird das Programm mit einer entsprechenden Meldung abgebrochen.
- Initialisierung des 8087 mit FINIT. Danach bekommt das Kontrollwort-Register (über FLDCW) den Wert $0372:

15							8	7							0
xx	xx	xx	IC	RC	RC	PC	PC	IEM	xx	PM	UM	OM	ZM	DM	IM
			0	0	0	1	1	0	?	1	1			1	

*) Der Name INIT87 ist (wie *StrCopy*, *StrNCopy* usw.) eine freie Erfindung von mir.

Überläufe (»Overflow«), Divisionen durch Null (»Zero«) und ungültige Befehle (»Invalid«) lösen einen Interrupt aus, der das Programm abbricht; Rundungsfehler (»Precision«), Unterläufe (»Underflow«) und denormalisierte Werte (»Denorm«) werden dagegen vom Coprozessor behandelt bzw. ignoriert. Rundungsrichtung (-> geradzahlige Werte) und Rechengenauigkeit (*Extended*) bleiben dieselben wie nach FINIT.

- Umsetzen der Interruptvektoren $02 und $75, deren Originalwerte bereits durch SYSTEM:@ gespeichert wurden.

Das Listing der Routine INIT87:

```
SYSTEM:INIT87 - Initialisierung des Coprozessors (V 4.0)
0302 DBE3        FINIT                      ; Initialisierung
0304 33C0        XOR    AX,AX
0306 A35B02      MOV    [025B],AX           ; DS:025B auf 00
0309 A35702      MOV    [0257],AX
030C D93E5B02    FSTCW  [025B]              ; Kontrollwort -> DS:025B
0310 B91400      MOV    CX,0014             ; WAIT geht hier nicht!
0313 E2FE        LOOP   0313
0315 A15B02      MOV    AX,[025B]           ; das Kontrollwort:
0318 253F0F      AND    AX,0F3F             ; xxxI RRPP xxPU OZDI
031B 3D3F03      CMP    AX,033F             ;        11   11 1111
031E 751B        JNZ    033B                ; --> kein Coprozessor!
```

Die Prüfung auf »Coprozessor installiert«: Weil der 8086 hier ewig auf das Signal »Operation beendet« warten würde, wenn kein Coprozessor installiert ist, steht hier anstelle des Befehls WAIT eine kurze Warteschleife. (Befehle wie FINIT oder FSTCW sind dagegen frei von unerwünschten Nebenwirkungen - falls kein 8087 installiert ist und dem 8086 »zuguckt«, bleiben sie einfach wirkungslos).

Der Befehl FINIT sollte sämtliche Ausnahmezustände maskieren (d.h. die entsprechenden Masken-Flags alle auf »1« setzen) und als Rechengenauigkeit *Extended* festlegen. Wenn das nachfolgende FSTCW nicht die erwartete Wirkung zeitigt, wird das Programm mit einer entsprechenden Meldung abgebrochen.

```
0320 9B          WAIT
0321 2ED92EFC02  FLDCW  CS:[02FC]           ; $0372 ->
0326 1E          PUSH   DS                    xxxI RRPP xxPU OZDI
0327 0E          PUSH   CS                           11  111   1
0328 1F          POP    DS
0329 BA7303      MOV    DX,0373             ; SetIntVec(2, SYSTEM:373)
032C B80225      MOV    AX,2502             ; *** NMI
032F CD21        INT    21
0331 BA6703      MOV    DX,0367             ; SetIntVec($75, SYSTEM:367)
0334 B87525      MOV    AX,2575             ; *** 8087-Fehlerbehandlung
0337 CD21        INT    21
0339 1F          POP    DS
033A CB          RETF
```

Nach dem Neusetzen des Kontrollwort-Registers werden die Interruptvektoren $02 und $75 auf Routinen innerhalb von *System* umgesetzt:

- Der Interrupt $02 (NMI) wird vom 8087 für nicht maskierte Ausnahmezustände erzeugt - hier also für Divisionen durch Null, Überläufe und ungültige Operationen. Die Behandlungsroutine in *System* prüft zuerst, ob der Interrupt tatsächlich durch den Coprozessor ausgelöst wurde - falls nicht, geht die Programmausführung nach einem Aufruf der Original-Behandlungsroutine von DOS (via *SaveIntVec02*) weiter.
- Die Routine für den Interrupt $75 speichert die Adresse des Befehls, der den Fehler ausgelöst hat, setzt den 8087 dann mit direkten Portbefehlen zurück und geht schließlich in die Behandlungsroutine für den Interrupt $02 über.

Ganz ehrlich: Wann und wo der Interrupt $75 ausgelöst wird, weiß ich nicht. In den Routinen von *System* habe ich jedenfalls vergeblich nach INT $75-Befehlen gesucht - genauso wie in verschiedenen Programmen mit recht komplexen Fließkomma-Operationen. An den Interrupt-Controllern programmiert Turbo Pascal erfreulicherweise nicht herum, d.h. ein hardwaremäßiges Auslösen dieses Interrupts scheint auch nicht in Frage zu kommen. Ob da in weiteren Versionen des Compilers noch etwas auf uns zukommt - vielleicht die vom IEEE-Komitee vorgeschlagene lokale Behandlungsroutine für Fließkommafehler?

Version 5.0

Die technischen Unterschiede für reine Pascal-Programmierer sind recht schnell abgehandelt:

- Der Emulator simuliert einen Coprozessor bis ins letzte Bit - nicht nur im Bezug auf Rechenbefehle, sondern auch, was Kontroll-, Status- und Tag-Register betrifft.
- Bei der Compilierung im Modus {$N+,E-} verhält sich ein 5.0-Programm auf Systemen ohne Coprozessor wie sein Gegenstück der Version 4.0 - es bricht mit einer entsprechenden Fehlermeldung ab.
- Bei der Compilierung im Modus {$N+,E+} wird - je nach Ausrüstung des Systems - entweder der Emulator oder der Coprozessor via FINIT initialisiert.
- Unabhängig davon, ob das Programm nun mit dem Emulator oder einem realen Coprozessor arbeitet, setzt INIT87 das Kontrollwort (via FLDCW) auf den Wert $1330:

15							8	7							0
xx	xx	xx	IC	RC	RC	PC	PC	IEM	xx	PM	UM	OM	ZM	DM	IM
			1	0	0	1	1	0	?	1	1				

Im Gegensatz zur Version 4.0 behandelt die Version 5.0 Unendlichkeiten als vorzeichenbehaftet und maskiert *DeNorm*-Ausnahmezustände *nicht*. Der erste Unterschied hat meines Wissens im praktischen Betrieb keine Auswirkungen - der zweite kuriert ein recht exotisches Problem, das vom folgenden Programm demonstriert wird:

```
program DeNormDemo;
{$N+}
type Double = Extended;     { <-- ausklammern }

var
  S: Single; D: Double;

begin
  S := 1.0E-36;
  repeat
    D := S;              { diese Double-Variable "läuft mit" }
    S := S / 2.0;
  until S = 0;

  Writeln('Kleinster als Single darstellbarer Wert: ', D);
end.
```

Eigentlich sollte dieses Programm den kleinsten Wert ermitteln, der sich (denormalisiert) im Format *Single* speichern läßt. Diese Funktion erfüllt es auch - es sei denn, man entfernt die Umdeklaration von *Double* in *Extended* und übersetzt es mit der Version 4.0 des Compilers. Was das Programm dann produziert, ist ein Laufzeitfehler: offensichtlich haben der 8087 und der 80287 bei der Konvertierung denormalisierter Werte in das Format *Double* Schwierigkeiten.

Die Version 5.0 überläßt die Behandlung von *DeNorm*-Ausnahmezuständen deshalb nicht dem Coprozessor, sondern definiert eine eigene Routine, auf die wir später noch einmal (zumindest ansatzweise) zurückkommen.

Der 80387 hat diese Probleme anscheinend nicht mehr: Wenn INIT87 einen Coprozessor dieses Typs erkennt, wird jedenfalls das EM-Bit im Kontrollwort wieder gesetzt (und die *DeNorm*-Behandlungsroutine damit überflüssig).

Ein Einschub zum Prinzip des 5.0-Emulators

hat an dieser Stelle zwar vom Thema her nicht das geringste zu suchen, scheint aber unvermeidlich: 8087-Befehle, Emulator-Operationen und die Initialisierung des Coprozessors sind in der Version 5.0 von Turbo Pascal so eng miteinander verknüpft, daß auch reine 8087-Programmierer nicht um die folgenden 10 Seiten herumkommen.

Fangen wir mit den wichtigsten Punkten an (die so phantastisch sind, daß mir nicht mehr als eine dürre Liste dazu eingefallen ist):

- Das Handbuch gibt an, daß sich der Compiler bei der Code-Erzeugung nicht sonderlich um den Stand des Schalters *O/C/Emulation* kümmert. Was dagegen nicht drin steht, ist eine fundierte Begründung, die in etwa so lauten könnte: direkte 8087-Befehle werden von Turbo Pascal *in keinem Fall* erzeugt - an die Stelle jeder Fließkomma-Operation tritt ein Verbund aus »WAIT« und »8087-Befehl«, zu dem jeweils eine von insgesamt 10 verschiedenen Konstanten addiert ist.

- Wie es der Zufall*) will, sind diese Konstanten so bemessen, daß sich für jede Kombination aus »WAIT«, »8087-Opcode« und »Konstante« ein Interrupt-Befehl (INT $34 .. INT $3D) zusammen mit einigen zusätzlichen Bytes ergibt. (Was auch bedeutet, daß DEBUG (im Gegensatz zu SYMDEB und CodeView) beim Listen eines mit Turbo Pascal erzeugten Fließkomma-Programms nur noch Unsinn liefert).

- Die entsprechenden Interruptvektoren werden durch INIT87 sämtlich auf denselben Dispatcher (»Funktionsverteiler«) gesetzt - auf eine Routine, die *den INT-Befehl selbst* durch Addition entsprechender »Gegenkonstanten« in die Befehlsfolge »WAIT« / »8087-Opcode« zurückverwandelt. Zusätzlich wird die (durch den Interrupt gespeicherte) RET-Adresse jeweils um 2 erniedrigt - mit dem Ergebnis, daß der Prozessor nach dem Rücksprung den soeben umgewandelten INT-Befehl als 8087-Opcode ausführt.

- Im Modus {$E+} nimmt der Linker einen zweiten Dispatcher mit auf, der anstelle einer Umwandlung von INT-Befehlen in 8087-Opcodes zu entsprechenden 8086-Unterprogrammen verzweigt (und den Code der aufrufenden Routine unverändert läßt). Auf diesen Dispatcher werden die Interruptvektoren $34..$3D durch die entsprechende Variante von INIT87 gesetzt, wenn die Prüfung des Systems mit dem Ergebnis »kein Coprozessor« endet.

  Im Modus {$E-} hat dieses Ergebnis den Abbruch des Programms mit der Meldung `Coprocessor required` zur Folge.

- Die Unterschiede zwischen den Modi {$E+} und {$E-} erschöpfen sich in separaten Varianten von INIT87 und den bei der Compilierung mit {$E+} aufgenommenen 8086-Unterprogrammen.

Allgemein: Wenn ein Coprozessor installiert ist, wandelt sich jeder Befehl der Art

```
CD3x yyyy zzzz      INT $3x yyyy zzzz...
```

sozusagen durch seine Ausführung in

```
9BDx yyyy zzzz      WAIT / 8087-Opcode yyyy zzzz...
```

*) Der Sinn das 8087, des Computers und vom ganzen Rest läßt sich mathematisch exakt berechnen. (Wie der an zeitgenössischer Literatur interessierte Leser natürlich längst weiß, lautet das Ergebnis 42).

um und bleibt danach für die restliche Laufzeit des Programms unverändert. Bei einer erneuten Ausführung (beispielsweise innerhalb einer Schleife) ist der Emulator nicht mehr beteiligt - er hat sich sozusagen selbst hinausgeworfen.

Diese »Einmal-Rückumwandlung läßt sich mit SYMDEB recht anschaulich demonstrieren. Gegeben sei ein Programm wie:

```
program EmuDemo;
{$N+,E-   Coprozessor vorausgesetzt}
const
  X: Extended = 3.0;
  Y: Extended = 4.0;
var
  Z: Extended;
begin
  Z := X + Y;
end.
```

Nach der Compilierung mit *O/L/Map file/Detailed* läßt sich das Ergebnis mit SYMDEB untersuchen:

```
mapsym emudemo                  <- erzeugt die Datei EMUDEMO.SYM
symdeb emudemo.sym emudemo.exe
-g a                            <- Initialisierung von SYSTEM, INIT87
-u @,21
EMUDEMO:@:
8: begin
4860:0000 9A00006348     CALL   4863:0000
4860:0005 9A61076348     CALL   4863:0761
4860:000A 55             PUSH   BP
4860:000B 89E5           MOV    BP,SP
9:        Z := X + Y;
4860:000D CD372E0A00     INT    37 ;FLD    TByte Ptr [X]
4860:0012 CD372E0000     INT    37 ;FLD    TByte Ptr [Y]
4860:0017 CD3AC1         INT    3A ;FADDP  ST(1),ST
4860:001A CD373E5000     INT    37 ;FSTP   TByte Ptr [Z]
4860:001F CD3D           INT    3D ;FWAIT
10:       end.
4860:0021 89EC           MOV    SP,BP
```

Der »Beweis«, daß die Umwandlung schrittweise stattfindet, könnte in etwa so aussehen:

```
-g 17                <- führt die ersten beiden Interrupt-Befehle aus
-u a,17
4860:000A 55             PUSH   BP
4860:000B 89E5           MOV    BP,SP
9:        Z := X + Y;
4860:000D 9B             WAIT
4860:000E DB2E0A00       FLD    TByte Ptr [Y]
```

```
4860:0012 9B          WAIT
4860:0013 DB2E0000    FLD    TByte Ptr [X]
4860:0017 CD3AC1      INT    3A ;FADDP  ST(1),ST
```

Und schließlich, nach der Ausführung bis zum **end** des Hauptprogramms:

```
-g 21
-u a,21
4860:000A 55          PUSH   BP
4860:000B 89E5        MOV    BP,SP
9:        Z := X + Y;
4860:000D 9B          WAIT
4860:000E DB2E0A00    FLD    TByte Ptr [Y]
4860:0012 9B          WAIT
4860:0013 DB2E0000    FLD    TByte Ptr [X]
4860:0017 9B          WAIT
4860:0018 DEC1        FADDP  ST(1),ST
4860:001A 9B          WAIT
4860:001B DB3E5000    FSTP   TByte Ptr [Z]
4860:001F 90          NOP
4860:0020 9B          WAIT
10:     end.
4860:0021 89EC        MOV        SP,BP
```

Zu guter Letzt - und vor einem Verweis auf den Anhang C, der sich mit den grauenvollen Bit-Tricksereien auseinandersetzt, aus denen die Umwandlung letztendlich besteht - noch einige weitere Behauptungen:

- Der 8087-Emulator von Turbo Pascal 5.0 entspricht zu 99,99% dem Emulator von Turbo C 2.0. Der Linker von Turbo C (TLINK) scheint allerdings über einige Fähigkeiten zu verfügen, die Turbo Pascal fehlen: Im Modus *O/C/C/ Floating point..8087* (= {$N+,E-}) werden die »Gegenkonstanten« durch TLINK addiert, die sich ergebende .EXE-Datei enthält also direkt ausführbare 8087-Befehle. Die 8087-Variante des Dispatchers wird allerdings auch hier in jedem Fall eingebunden.
- Die Routine INIT87 belegt einen weiteren Interrupt ($3E), der von Borland (bei Turbo C) als *Shortcut* bezeichnet wird und insgesamt 18 Subfunktionen zusammenfaßt, von denen allerdings nur fünf durch die Laufzeitbibliothek benutzt werden. (Weiteres dazu finden Sie ebenfalls in Anhang C).

Die beiden Versionen von INIT87

In einem mit {$N+} compilierten Programm wird INIT87 wie in der Version 4.0 direkt nach dem Initialisierungsteil von *System* aufgerufen und besteht aus drei Schritten:

1. Prüfung des Systems auf »Coprozessor vorhanden« über den Aufruf eines Unterprogramms (das seinerseits die Variable *Test8087* setzt).

2a. Wenn die Prüfung positiv verlaufen ist, werden die Startadressen der beiden 8087-Dispatcher in zwei Register geladen.

2b. Falls die Prüfung negativ verläuft, erhöht die Routine die globale Variable *StackLimit* von 0 auf 230 Bytes und lädt die Startadressen der 8086-Dispatcher in zwei Register; die {$E-}-Variante von INIT87 bricht das Programm mit der Meldung `Coprocessor required` über einen Aufruf von *Halt* ab.

3. Sprung zu einer Routine, die die Interruptvektoren $34..$3D und $3E sowie $02 und $75 setzt, den 8087 initialisiert und schließlich zum Pascal-Hauptprogramm zurückkehrt.

Nach diesen Erläuterungen sollte das Listing der {$E+}-Variante eigentlich selbsterklärend sein:

```
SYSTEM:INIT87 im Modus {$N+,E+}
0761 E8AFFA          CALL   0213                      ; 8087 vorhanden?
0764 BE5F30          MOV    SI,305F                   ; $34..$3D -> SYSTEM:305F
0767 BFDF30          MOV    DI,30DF                   ; INT $3E -> SYSTEM:30DF
076A 0AC0            OR     AL,AL
076C 750C            JNZ    077A                      ; -> Coprozessor installiert

- kein 8087 im System
076E C7064600E600    MOV    Word Ptr [STACKLIMIT],00E6 ; Stacklimit rauf
0774 BE2428          MOV    SI,2824                   ; $34..$3D -> SYSTEM:2824
0777 BFF424          MOV    DI,24F4                   ; INT $3E -> SYSTEM:24F4

077A E91DFB          JMP    029A                      ; Setzen der Interrupts
```

Das Listing der {$E-}-Variante ist auch nicht länger:

```
SYSTEM:INIT87 im Modus {$N+,E-}
0761 E8AFFA          CALL   0213                      ; Prüfung "8087 vorhanden"
0764 0AC0            OR     AL,AL                     ; (Ergebnis in AL)
0766 7409            JZ     0771                      ; kein 8087!

0768 BE880A          MOV    SI,0A88                   ; $34..$3D -> SYSTEM:0A88
076B BF080B          MOV    DI,0B08                   ; INT $3E -> SYSTEM:0B08
076E E929FB          JMP    029A                      ; Vektoren setzen, 8087-Init

- kein Coprozessor vorhanden
0771 BB7D07          MOV    BX,077D                   ; "Coprocessor required"
0774 E81DFA          CALL   0194                      ; Ausgabe der Meldung
0777 B8FF00          MOV    AX,00FF                   ; Exitcode 255
077A E95BF9          JMP    HALT                      ; Abbruch
```

Falls Sie sich über den Exitcode 255 wundern, der a) in Kapitel 1 als undokumentiert bezeichnet wurde und b) eigentlich für den Programmabbruch via CONTROL-BREAK stehen sollte, haben Sie recht: offensichtlich hat hier die Kommunikation zwischen den Entwicklern einerseits und dem Handbuchautor andererseits nicht so hundertprozentig funktioniert.

Die Prüfung des Systems

geschieht - genauso wie alle weiteren Schritte der 8087-Initialisierung - unabhängig vom Stand des Schalters {$E} bei der Compilierung (und in beiden Varianten der Laufzeitbibliothek mit denselben Adressen). Die ausgeführten Schritte lassen sich folgendermaßen charakterisieren:

1. Absuchen der *Environment*-Tabelle nach einem Eintrag der Form `87=...` Falls die Suche erfolgreich ist, wird die direkte Prüfung des 8087 in jedem Fall übersprungen. Bleibt sie ergebnislos, folgen - auch für den Modus {$E+} - einige direkte 8087-Befehle.
2. Wenn ein Coprozessor installiert ist: Unterscheidung zwischen 8087 und 80287 über eine Prüfung des 8086 - die Routine geht davon aus, daß auf AT-Systemen kein 8087 installiert sein kann und umgekehrt*). Für Systeme der Klasse AT und höher folgt schließlich eine Unterscheidung zwischen 80287 und 80387, die wiederum über 8087-Befehle läuft.
3. Mit dem Ergebnis der Prüfungen wird zum einen die Variable *Test8087*, zum anderen das spätere Kontrollwort für den Coprozessor gesetzt.

Das Listing:

```
- Prüfung auf "Coprozessor vorhanden"
0213 BBA402        MOV   BX,02A4              ; für FSTCW
0216 33FF          XOR   DI,DI
0218 8E064400      MOV   ES,[PREFIXSEG]       ; ES:DI = Prefix:002C
021C 268E062C00    MOV   ES,ES:[002C]           = Start der Environment-
0221 B9FF7F        MOV   CX,7FFF                Tabelle. Max. Länge 32K
0224 FC            CLD

- Absuchen der Environment-Tabelle nach "87=Y" oder "87="N"
0225 268B05        MOV   AX,ES:[DI]           ; ein Byte der Tabelle
0228 0AC0          OR    AL,AL                ; Tabellenende? (2. NUL?)
022A 741B          JZ    0247                 ; -> ja
022C 3D3837        CMP   AX,3738              ; "87" (rückwärts)?
022F 7510          JNZ   0241                   -> nein
0231 268B4502      MOV   AX,ES:[DI+02]        ; "87" gefunden.
0235 3C3D          CMP   AL,3D                  Folgt ein "="?
0237 7508          JNZ   0241                   -> nein
```

*) Wenn das mal immer gutgeht... Wie sieht es eigentlich mit 80186-Systemen aus?

```
0239 80E4DF        AND    AH,DF              ; (-> Großbuchstaben)
023C 80FC59        CMP    AH,59              ;'Y' ?
023F EB1D          JMP    025E               ; 8087-Test überspringen(!)
0241 33C0          XOR    AX,AX
0243 F2            REPNZ                     ; Suche nach dem Ende
0244 AE            SCASB                       des mom. Eintrags
0245 74DE          JZ     0225               ; nächster Eintrag
```

Wie in Kapitel 1 (und im Referenzhandbuch) beschrieben, besteht die Tabelle *Environment* aus einer unsortierten Folge von ASCIIZ-Strings; der letzte Eintrag hat die Länge 0 (d.h. besteht nur aus einem NUL-Zeichen).

Ein eher nebensächliches Detail: Wenn die Routine einen Eintrag `87=` gefunden hat, vergleicht sie lediglich das nächste Zeichen mit `Y` - eine Environment-Variable wie `87=YXZ` wird also als »8087 vorhanden« gewertet; jedes Zeichen außer `Y` nach `87=` führt zu einem negativen Ergebnis der Prüfung*).

```
0247 33C0          XOR    AX,AX
0249 E6F0          OUT    F0,AL              ; ERROR-Leitung zurück
024B DBE3          FINIT                     ; Initialisierung
024D 8907          MOV    [BX],AX            ; Platz für CW auf 0
024F D93F          FSTCW  [BX]               ; und Speichern des CW
0251 B91400        MOV    CX,0014            ; WAIT geht hier nicht!
0254 E2FE          LOOP   0254
0256 8B07          MOV    AX,[BX]            ; das Kontrollwort:
0258 253F0F        AND    AX,0F3F            ; xxxI RRPP xxPU OZDI
025B 3D3F03        CMP    AX,033F            ;        11   11 1111
025E BA3013        MOV    DX,1330            ; späteres Kontrollwort
0261 B000          MOV    AL,00
0263 752D          JNZ    0292               ; kein 8087 -> Test8087 := 0
```

Nach Zurücksetzen der ERROR-Leitung des 8087 über einen direkten Portzugriff findet dieselbe Prüfung des 8087 wie in der Version 4.0 statt (wobei hier allerdings gleich noch der spätere Wert des Kontrollwort-Registers für 8088-Maschinen in das DX-Register geladen wird).

```
- Prüfung des Prozessors: 8088 oder 80286/386?
0265 54            PUSH   SP              <- PUSH "SP-02" für 8088,
0266 58            POP    AX                     "SP-00" für 80286/386
0267 3BC4          CMP    AX,SP
0269 B001          MOV    AL,01
026B 7525          JNZ    0292               ; ist 8088 -> Test8087 := 1
```

Die Prüfung des 8086 findet nur statt, wenn die vorangehenden 8087-Operationen erfolgreich verlaufen sind, und macht sich eine der wenigen Imkompatibilitäten dieser Prozessorfamilie zunutze: der 8088 und der 8086 erniedrigen SP beim

*) Dasselbe gilt übrigens auch für sämtliche Versionen von Turbo C sowie für Turbo Basic.

Befehl `PUSH SP` *vor* der Speicherung, die restlichen Prozessorversionen dagegen danach.

Für 8088- bzw. 8086-Maschinen (sprich: PCs und XTs) nimmt die Routine an, daß ein 8087 installiert ist, und verzichtet auf den folgenden Test: die Variable *Test8087* bekommt den Wert 1, das (später gesetzte) Kontrollwort den Wert $1330.

```
- Prüfung des Coprozessors: 80287 oder 80387?
026E DBE3          FINIT        *** WAIT-Befehle herausgekürzt!
0271 D9E8          FLD1                          ; 1.0
0274 D9EE          FLDZ                          ; 0.0
0277 DEF9          FDIVP  ST(1),ST               ; 1.0/0.0, ergibt Unendlich
027A D9C0          FLD    ST(0)                  ; Ergebnis verdoppeln
027D D9E0          FCHS                          ; und Vorzeichen umdrehen
0280 DED9          FCOMPP ST(1)                  ; beide Werte "gleich"?
0283 DD3F          FSTSW  [BX]
0286 8B07          MOV    AX,[BX]
0288 9E            SAHF                          ; Ergebnis -> 8086-Flags
0289 B002          MOV    AL,02
028B 7405          JZ     0292                   ; gleich: ist 80287
028D BA3213        MOV    DX,1332
0290 B003          MOV    AL,03                  ; ungleich: 80387, CW = 1332
```

Die Unterscheidung zwischen 80287 und 80387: der 80287 setzt beim Befehl FINIT das projektive Modell (d.h. behandelt Unendlichkeiten vorzeichenlos), der 80387 arbeitet dagegen mit dem affinen Modell - hier wird der Unterschied zwischen *-Inf* und *+Inf* erkannt*). Für einen 80387 wird der spätere Wert des Kontrollwortes auf $1332 gesetzt (vgl. Seite 107/108), für einen 80287 bleibt es beim Wert $1330.

```
- Setzen von Test8087 und des späteren Kontrollwortes
0292 A24F00        MOV    [TEST8087],AL          ; 0/1/2/3
0295 8916A202      MOV    [SAVEINT75+04],DX      ; $1330 für 80(2)87,
0299 C3            RET                             $1332 für 80387
```

Das abschließende RET führt zum zweiten Befehl von INIT87 zurück.

... und das Setzen der Vektoren

Nachdem INIT87 abhängig vom Ergebnis der Prüfung auf »8087 vorhanden« die Register SI und DI mit den Startadressen der Dispatcher für $34..$3D bzw. $3E gesetzt hat, geht es mit einem Sprung SYSTEM:029A weiter. Code und Adresslage dieser Routine sind ebenfalls vom Stand des Schalters {$E} unabhängig - das einzige, was sich je nach Ausrüstung des Systems ändert, sind die Werte von SI und DI:

*) Bitte nicht vergessen: nach dem Befehl FINIT sind alle Ausnahmezustände maskiert - nur deshalb führt die Division durch 0.0 nicht zu einem Interrupt. Und noch ein wenig Besserwisserei dazu: nachdem die Routine an diesem Punkt sowieso von einem 80287 bzw. 80387 ausgeht, hätte man das Vergleichsergebnis auch mit einem direkten Befehl in AX schreiben können.

```
- Setzen der Vektoren und Initialisierung des Coprozessors
  Aufruf mit SI = Offset für Vektoren $34..$3D (im Segment SYSTEM)
             DI = Offset für Vektor $3E (ebenfalls im Segment SYSTEM)
029A 1E          PUSH  DS
029B 0E          PUSH  CS
029C 1F          POP   DS                    ; DS aufs Codesegment
029D B83425      MOV   AX,2534               ; "Set Interrupt $34"
02A0 B90A00      MOV   CX,000A               ; $0A Vektoren ($34..$3D)
02A3 8BD6        MOV   DX,SI                 ; (neue Adresse in DS:DX)
02A5 CD21        INT   21                    ; Vektor setzen
02A7 40          INC   AX                    ; nächste Vektornummer
02A8 E2FB        LOOP  02A5

02AA 8BD7        MOV   DX,DI                 ; Adresse für Vektor $3F
02AC CD21        INT   21                    ; und Setzen
```

Da der Initialisierungsteil von *System* bereits die Originalwerte der Vektoren $34..$3D und $3E gespeichert hat, werden diese Interrupts hier ohne weitere Vorsichtsmaßnahmen verändert. Dasselbe gilt für die Vektoren $02 und $75:

```
02AE BAD202      MOV   DX,02D2
02B1 B002        MOV   AL,02                 ; INT $02 -> SYSTEM:02D2
02B3 CD21        INT   21
02B5 BAC602      MOV   DX,02C6
02B8 B075        MOV   AL,75                 ; INT $75 -> SYSTEM:02C6
02BA CD21        INT   21
02BC 1F          POP   DS                    ; DS wieder aufs DSeg
```

Die »endgültige« Initialisierung des 8087 findet über Emulator-Interrupts statt - wie gesagt, ist auch dieser Teil von INIT87 in beiden Versionen identisch.

```
02BD CD37E3      INT   37 ; FINIT            ; Initialisierung
02C0 CD352EA202  INT   35 ; FLDCW [SAVEINT75+04]   ; (von der Prüfungs-
02C5 CB          RETF                          routine ($0293) gesetzt)
```

Die Behandlung von 8087-Fehlern

ist in der Version 5.0 größtenteils nach demselben Muster wie in der Version 4.0 aufgebaut - auch hier setzt die Behandlungsroutine für den Interrupt $75 zuerst den Coprozessor durch eine Reihe direkter Portbefehle zurück*), um dann in die Routine für den Interrupt $02 überzugehen:

```
- Hierher zeigt der Vektor $75 nach der Initialisierung
02C6 50          PUSH  AX
02C7 32C0        XOR   AL,AL
02C9 E6F0        OUT   F0,AL                 ; ERROR-Leitung zurück
02CB B020        MOV   AL,20
```

*) Was bei einem System ohne Coprozessor anscheinend nichts schadet - oder kann der Interrupt $75 in diesem Fall überhaupt nicht erzeugt werden?

```
02CD E6A0          OUT    A0,AL                  ; .. soso...
02CF E620          OUT    20,AL                  ; (dito)
02D1 58            POP    AX                     ; weiter mit INT $02
```

Die Routine für den Interrupt $02 enthält dagegen einige echte Neuerungen:

```
- Hierher zeigt der Vektor $02 nach der Initialisierung
02D2 50            PUSH   AX
02D3 1E            PUSH   DS
02D4 B82549        MOV    AX,DATA                ; DS aufs DSeg
02D7 8ED8          MOV    DS,AX
02D9 803E4F0000    CMP    Byte Ptr [TEST8087],00 ; 8087 installiert?
02DE 7507          JNZ    02E7                   ; -> ja
02E0 CD3536A802    INT    35 ; FSTENV [02A8]     ; nein -> EMU-Status
02E5 EB05          JMP    02EC                     speichern

02E7 D936A802      FSTENV [02A8]                 ; 8087-Status speichern
02EB 9B            WAIT
```

Hier scheint einmal eine Trennung zwischen »Emulator« und »8087« geplant gewesen zu sein, die mittlerweile wieder unter den Tisch gefallen ist[1].

```
02EC A0A802        MOV    AL,[02A8]              ; Kontrollwort, B7..B0
02EF F6D0          NOT    AL                     ; (NOT IEM)
02F1 2206AA02      AND    AL,[02AA]              ; AND IR (Statuswort, B7)
02F5 7808          JS     02FF                   ; -> 8087-Interrupt
```

Die Prüfung, ob der Interrupt tatsächlich vom Coprozessor ausgelöst wurde, geschieht über die Bits IEM und IR des Kontroll- bzw. Statuswortes: Wenn IEM *nicht* gesetzt ist und der 8087 über IR signalisiert, daß er den Grund für die Unterbrechung des 8086 darstellt, dann geht es mit einem Sprung zu $2FF weiter.

```
- INT $02 nicht durch 8087 bzw. Emulator ausgelöst: Aufruf Originalroutine
02F7 9C            PUSHF
02F8 FF1E5E02      CALL   FAR [SAVEINT02]        ; Originalvektor $02
02FC 1F            POP    DS
02FD 58            POP    AX
02FE CF            IRET
```

Ansonsten folgt hier der Aufruf über den Originalvektor $02 (und damit normalerweise ein Sprung zu DOS, wobei ich mir nicht darüber im klaren bin, ob das anschließende IRET mehr eine Formsache darstellt: zumindest nach einem Paritätsfehler ist dieser Aufruf eine Einbahnstraße - und andere Auslöser für den INT $02 sind mir nicht bekannt)[2].

---

[1] In den Turbo C-Versionen 1.5 und 2.0 enthalten die beiden Zweige tatsächlich unterschiedliche Befehle.

[2] Auf gut Deutsch: Was diesen Punkt betrifft, bin ich im letzten Jahr nicht schlauer geworden.

```
- INT $02 durch 8087 bzw. Emulator ausgelöst: TURBO-Fehlerbehandlung
02FF FB          STI
0300 A83D        TEST   AL,3D              ; 0011 1101 DeNorm?
0302 743F        JZ     0343               ; -> ja, direkte Korrektur
```

Falls der Interrupt durch eine 8087-Operation mit denormalisierten Zahlen ausgelöst wurde (was bei Verwendung des 80387 nie der Fall ist - dort ist der entsprechende Ausnahmezustand durch den Wert $1332 im Kontrollwort grundsätzlich maskiert), dann folgt hier ein Sprung zu einer internen Korrekturroutine*).

```
0304 CD37E3      INT    37 ;FINIT
0307 CD352EA202  INT    35 ;FLDCW  [02A2]
030C 59          POP    CX                 ; DS vom Stack
030D 59          POP    CX                 ; AX  ...
030E 59          POP    CX                 ; Flags  ...
030F 5B          POP    BX                 ; IRET-Adresse (Segment)
0310 81FB6948    CMP    BX,4869            ; SYSTEM-Segment?
0314 7503        JNZ    0319

0316 59          POP    CX                 ; IRET-Adresse (Offset)
0317 59          POP    CX                 ; (nochmal 4 Bytes
0318 5B          POP    BX                   vom Stack)
```

In allen anderen Fällen geht es mit der Ermittlung der Fehleradresse weiter: Falls sich die Probleme innerhalb der Laufzeitbibliothek ergeben haben, holt das Programm eine weitere RET-Adresse vom Stack, damit die Fehlermeldung den Stand von CS:IP innerhalb des Programms angibt (und nicht innerhalb der Bibliotheksroutine selbst).

Was nun noch vor dem endgültigen Abbruch des Programms folgt, ist die Festlegung des Exit-Codes anhand der 8087-Statusflags:

```
0319 A801        TEST   AL,01
031B 7515        JNZ    0332

031D BAC800      MOV    DX,00C8            ; Laufzeitfehler 200
0320 A804        TEST   AL,04                (Division durch 0)
0322 7511        JNZ    0335

0324 BACD00      MOV    DX,00CD            ; Laufzeitfehler 205
0327 A808        TEST   AL,08                (Fließkomma-Überlauf)
0329 750A        JNZ    0335
```

*) ... die weder in diesem Kapitel noch in Anhang C gelistet ist: Soweit sich feststellen ließ, spielen bei der notwendigen Format-Konvertierung einige Routinen des ROM-BIOS(!), die Original-Fehlerbehandlung von DOS, unzählige Prüfungen des 8087-Environments und nicht zuletzt die Unterscheidung zwischen 8087 und 80287 (via *Test8087*) eine Rolle. Wir beschränken uns deshalb auf die lapidare Feststellung, daß die Sache im gewünschten Sinne zu funktionieren scheint.

```
032B BACE00        MOV    DX,00CE              ; Laufzeitfehler 206
032E A810          TEST   AL,10                  (Fließkomma-Unterlauf)
0330 7503          JNZ    0335

0332 BACF00        MOV    DX,00CF              ; 207: unspezifisch

0335 92            XCHG   AX,DX                ; Exit-Code -> AX
0336 E9A3FD        JMP    00DC                 ; Programm-Abbruch via HALT
```

Die »offizielle« Startadresse von *Halt* ist SYSTEM:00D8 - die ersten beiden Befehle dieser Routine (die den Wert 0000:0000 in CX/BX für die spätere Speicherung in *ErrorAddr* setzen), werden hier umgangen.

Realzahlen als Parameter

Wie in Kapitel 1 angekündigt (und dort aus fast allen Beispielen ausgeklammert), verwendet Turbo Pascal die Register des 8087 zur Übergabe von Parametern und Funktionsergebnissen der Typen *Single*, *Double*, *Extended* und *Comp*.

Fangen wir auch hier mit einer Prozedur an, die einen einzigen Parameter erwartet, und nehmen wir gleich eine lokale Variable dazu:

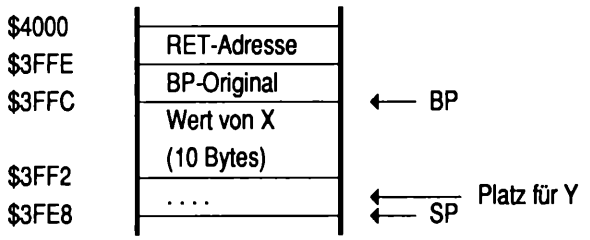
```
program FloatProc;
{$N+}
var
   A: Extended;

procedure Proc1(X: Extended);
var Y: Extended;
begin
  Y:= X;
end;

begin
  A := 4.349587;
  Proc1(A);
end.
```

Das Ergebnis der Übersetzung durch den Compiler in der Version 4.0:

```
FLOATPROC:PROC1:
** begin
0000 55            PUSH   BP
0001 89E5          MOV    BP,SP
0003 83EC14        SUB    SP,+14               ; 20 Bytes lokale Variablen
0006 9B            WAIT
0007 DB7EF6        FSTP   TByte Ptr [BP-0A]    ; der übergebene Parameter X
000A 9B            WAIT
```

```
** Y:= X;
000B 9B            WAIT
000C DB6EF6        FLD    TByte Ptr [BP-0A]     ; X auf den 8087-Stack
000F 9B            WAIT
0010 DB7EEC        FSTP   TByte Ptr [BP-14]     ; Speicherung in Y
0013 9B            WAIT
** end;
0014 89EC          MOV    SP,BP                 ; Lokales vom Stack
0016 5D            POP    BP
0017 C3            RET                          ; <== keine Parameter!
```

Der Compiler hat hier (ähnlich wie bei strukturierten Wertparametern) zusätzliche Befehle für »Lokalkopien« im Startcode der Prozedur erzeugt. Der Stack innerhalb von *Proc1* vor der Zuweisung *Y := X*:



Der als *X* übergebene Wert wird als allererstes wieder vom 8087-Stack geholt und lokal (d.h. auf dem 8086-Stack) gespeichert. Weil diese Speicherung mit einem »Pop« verbunden ist, hat das TOS-Register des 8087 danach keinen definierten Inhalt mehr und muß für die Zuweisung *X:= Y;* erst wieder mit dem Wert von *X* geladen werden*).

```
0018  2C 83 6A 83 13 D1 2F 8B 01 40  +0.4349587E+1

FLOATPROC:@:
** begin
0022 9A0000663A    CALL   SYSTEM:@              ; Initialisierung
0027 9AAF02663A    CALL   SYSTEM:INIT87         ; Init 8087
002C 89E5          MOV    BP,SP

** A := 4.349587;
002E 9B            WAIT
002F 2EDB2E1800    FLD    TByte Ptr CS:[0018]   ; Laden der Konstanten
0034 9B            WAIT
0035 DB3E0000      FSTP   TByte Ptr [A]         ; Speicherung in A
0039 9B            WAIT
```

*) Das sieht umständlich aus, ist aber prinzipbedingt: der Compiler kann nicht wissen, ob das Programm auf übergebene Parameter direkt in der nächsten Anweisung zugreift. Weil sich ein Parameter mehrfach in einer Routine verwenden läßt, ist eine »Lokalkopie« immer notwendig - eine »Vorausanalyse« der folgenden Anweisung könnte allerdings unnötige »Pops« und spätere Ladeaktionen vermeiden.

Untypisierte Konstanten der Typen *Single..Extended* werden (wie untypisierte Strings) direkt vor der jeweiligen Routine im Codesegment gespeichert. Die Zuweisung an die Variable *A* arbeitet nach demselben Prinzip wie die Zuweisung *X := Y* in *Proc1*: der Wert wird ins TOS-Register des 8087 geladen und von dort aus in die Zielvariable geschrieben.

```
** Proc1(A);
003A 9B            WAIT
003B DB2E0000      FLD    TByte Ptr [A]        ; Wert A auf den 8087-Stack
003F E8BEFF        CALL   PROC1                ; Aufruf
** end.
0042 89EC          MOV    SP,BP
0044 31C0          XOR    AX,AX                ; Exitcode 0
0046 9AF301663A    CALL   SYSTEM:HALT          ; Programmende
```

Vor dem Aufruf von *Proc1* wird der Wert von *A* ins TOS-Register geladen, nach dem Aufruf geht es normal im Programm weiter: Wie bei »normalen« Übergaben ist die aufgerufene Routine für das Entfernen der Parameter verantwortlich.

Version 5.0

Hier haben wir den einzigen programmiertechnischen Unterschied, dessen Dokumentation sich in aller Ausführlichkeit im Handbuch findet - was auch so sein muß: Existierende 4.0-Programme, die via **inline** oder über Maschinensprache-Routinen direkt auf Fließkomma-Parameter zugreifen, finden ohne entsprechende Modifikationen ein schnelles Ende.

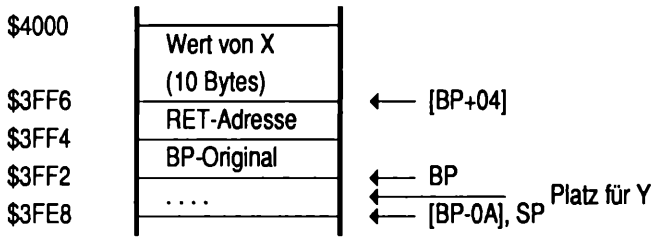
Die 5.0-Version unseres Beispiels *Proc1* sieht (nach ihrer Ausführung auf einem System mit installiertem Coprozessor, vgl. Seite 94) so aus:

```
FLOATPROC:PROC1:
** begin
0000 55            PUSH   BP
0001 89E5          MOV    BP,SP
0003 83EC0A        SUB    SP,+0A               ; 10 Bytes Platz für Y
** Y := X;
0006 9B            WAIT
0007 DB6E04        FLD    TByte Ptr [BP+04]    ; X vom 8086-Stack -> TOS
000A 9B            WAIT
000B DB7EF6        FSTP   TByte Ptr [BP-0A]    ; Speicherung in Y
000E 90            NOP
000F 9B            WAIT
** end;
0010 89EC          MOV    SP,BP                ; Lokales (d.h. Y) vom Stack
0012 5D            POP    BP
0013 C20A00        RET    000A                 ; Parameter X vom Stack
```

Hier werden Fließkomma-Parameter also nicht als strukturiert, sondern als einzelner Wert behandelt. Dementsprechend

- findet die Übergabe durch den 8086-Stack statt
- entfällt die Notwendigkeit von »Lokalkopien«
- gibt die Routine den für den Parameter *X* belegten Platz auf dem 8086-Stack über den Befehl RET $000A wieder frei.

Zum Vergleich auch hier der Stack innerhalb von *Proc1* direkt vor der Zuweisung *Y := X*:



Für die Arbeit mit **inline**-Code innerhalb einer Pascal-Routine ließen sich die Unterschiede recht griffig zusammenfassen: »Während der Zugriff auf Fließkomma-Wertparameter in der Version 4.0 über negative Offsets zu BP stattfindet, verwendet die Version 5.0 positive Offsets - die Parameter liegen hier also oberhalb der RET-Adresse auf dem Stack.«

Maschinenroutinen, die Fließkomma-Wertparameter erwarten und gleich über die 8087-Register weiterverarbeiten, müssen dagegen die entsprechenden Ladebefehle »zu Fuß« nachholen - wir kommen im Abschnitt über die direkte Programmierung des 8087 noch darauf zurück*).

Das Hauptprogramm von *FloatProc* reflektiert die veränderte Parameterübergabe (und ist wieder *nach* seiner Ausführung gelistet):

```
0016  26 68 82 97 CF A8 2F 8B 01 40  +0.43495678E+1

FLOATPROC:@:
** begin
0020 9A00006648     CALL   SYSTEM:@              ; Initialisierung
0025 9A61076648     CALL   SYSTEM:INIT87         ; Init 8087
002A 55             PUSH   BP
002B 89E5           MOV    BP,SP
```

*) Wo bleibt die Meckerei? Nirgends - die Behandlung von Fließkomma-Wertparametern durch die Version 5.0 stellt eine sehr konsequente und geradlinige Erweiterung »normaler« Wert-Übergaben dar und macht beispielsweise die Umstellung eigener Routinen von *Real* auf *Extended* wesentlich einfacher.

```
** A := 4.3495678;
002D 9B            WAIT
002E 2EDB2E1600    FLD    TByte Ptr CS:[0016]    ; Laden der Konstanten
0033 9B            WAIT
0034 DB3E3C00      FSTP   TByte Ptr [A]          ; Speicherung in A
0038 90            NOP
0039 9B            WAIT
```

Untypisierte Konstanten der Formate *Single..Extended* werden auch in der Version 5.0 direkt vor der entsprechenden Routine im Codesegment gespeichert; die Zuweisung an *A* zeigt ebenfalls keine Neuerungen (abgesehen von dem zusätzlichen NOP vor dem letzten WAIT, an dem das Prinzip des Emulators schuld ist).

```
** Proc1(A);
003A FF364400      PUSH   [A+08]                 ; Wert von A mit 8086-
003E FF364200      PUSH   [A+06]                   Befehlen auf den
0042 FF364000      PUSH   [A+04]                   8086-Stack
0046 FF363E00      PUSH   [A+02]
004A FF363C00      PUSH   [A]
004E E8AFFF        CALL   PROC1                  ; Aufruf
```

Der »Beweis«, daß die Übergabe von Fließkommazahlen den 8086-Stack verwendet. Wie zu sehen, kommt die Version 5.0 beim Vergleich der Codegröße ausgesprochen schlecht weg*) - hier stehen 20 Bytes für die schrittweise Kopie des Wertes von *A* den fünf Bytes der Version 4.0 (für `WAIT / FLD TByte Ptr[A]`) gegenüber. (Der gezeigte Aufruf von *Proc1* stellt allerdings einen Extremfall dar - die Wertübergabe von *Double* kommt mit 16, die Übergabe von *Single* mit 8 Bytes aus).

```
** end.
0051 89EC          MOV    SP,BP
0053 5D            POP    BP
......             .....
```

Das Ende der 5.0-Version von *FloatProc* zeigt (bis auf das unvermeidliche POP BP) keine weiteren Unterschiede.

Mehrere und »gemischte« Parameter

Beim Aufruf einer Routine bringt der Compiler die zu übergebenden Parameter der Reihe nach auf den Stack - nur daß es sich in der Version 4.0 eben um zwei verschiedene Stacks handeln kann. Der Aufruf einer als

```
procedure Proc2(X: Extended; Y: Integer; Z: Single);
```

deklarierten Routine wird vom Compiler in der Version 4.0 folgendermaßen übersetzt:

*) In puncto Geschwindigkeit hat die Version 5.0 allerdings in den meisten Fällen die Nase vorn - wir kommen im Abschnitt über Benchmarks noch darauf zurück.

```
   ....
** Proc2(4.0,6,2.0);
004F 2EDB2E2E00     FLD    TByte Ptr CS:[001E]    ; Konstante 4.0 -> 8087
0054 B80600         MOV    AX,0006
0057 50             PUSH   AX                     ; Konstante 6 -> 8086-Stack
0059 2EDB2E3800     FLD    TByte Ptr CS:[0028]    ; Konstante 2.0 -> 8087
005E E89FFF         CALL   PROC2                  ; Aufruf
```

Ein **Hinweis:** Aus diesem Listing wurden (wie auch in den folgenden Besprechungen) alle WAIT-Befehle herausgekürzt.

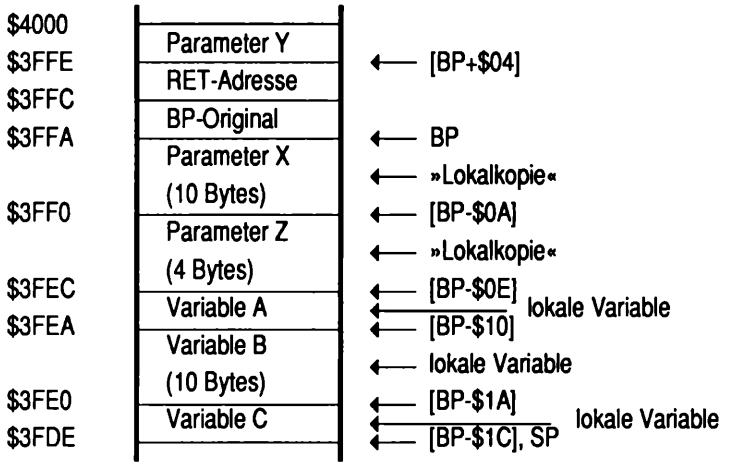
Innerhalb der Routine reserviert der Compiler (wie bei »Lokalkopien« strukturierter Wertparameter) *zuerst* Platz für die übergebenen Realzahlen, danach kommen lokal deklarierte Variablen. Ein vollständiges Beispiel:

```
procedure Proc2(X: Extended; Y: Integer; Z: Single);
var
  A: Integer;
  B: Extended;
  C: Integer;
begin
  B := X + Z;
  A := Y;
  C := A;
end;
```

Die Übersetzung von *Proc2* durch den Compiler in der Version 4.0:

```
FLOATPROC:PROC2:   *** WAIT-Befehle herausgekürzt
** begin
0000 55             PUSH   BP
0001 89E5           MOV    BP,SP                  ; 14 Bytes Lokalkopien,
0003 83EC1C         SUB    SP,+1C                   14 Bytes lokale Variablen
0007 D95EF2         FSTP   DWord Ptr [BP-0E]      ; Parameter Z (Single)
000B DB7EF6         FSTP   TByte Ptr [BP-0A]      ; Parameter X
** B := X + Z;
0010 DB6EF6         FLD    TByte Ptr [BP-0A]      ; X -> TOS
0014 D846F2         FADD   DWord Ptr [BP-0E]      ; Z dazu
0018 DB7EE6         FSTP   TByte Ptr [BP-1A]      ; -> Variable B
** A := Y;
001C 8B4604         MOV    AX,[BP+04]             ; Parameter Y -> AX
001F 8946F0         MOV    [BP-10],AX             ; -> Variable A
** C := A;
0022 8B46F0         MOV    AX,[BP-10]             ; Variable A -> AX
0025 8946E4         MOV    [BP-1C],AX             ; -> Variable C
** end;
0028 89EC           MOV    SP,BP                  ; Lokales vom Stack
002A 5D             POP    BP
002B C20200         RET    0002                   ; Parameter Y vom Stack
```

Und der Stack innerhalb von *Proc2*:



Version 5.0

Wie bereits in einer Fußnote angemerkt, stellt die Übergabe von Fließkommawerten durch den 8086-Stack eine konsequente und geradlinige Erweiterung dar. Der Aufruf von *Proc2* hat also im Prinzip dieselbe Struktur wie der Aufruf einer x-beliebigen Routine mit unstrukturierten Wertparametern:

```
   ....
** Proc2(4.0, 6, 2.0);     *** nach Ausführung, WAIT-Befehle herausgekürzt
0047 2EFF362E00     PUSH   CS:[001B]              ; Konstante 4.0
004C 2EFF362C00     PUSH   CS:[0019]                = Parameter X
0051 2EFF362A00     PUSH   CS:[0017]
0056 2EFF362800     PUSH   CS:[0015]
005B 2EFF362600     PUSH   CS:[0013]
0060 B80600         MOV    AX,0006                ; Parameter Y
0063 50             PUSH   AX
0065 2EDB2E3000     FLD    TByte Ptr CS:[001D]     ; als EXTENDED gespeichert:
006A 83EC04         SUB    SP,+04                  ; 4 Bytes Platz auf
006D 89E3           MOV    BX,SP                     dem Stack, BX als Zeiger
0070 36D91F         FSTP   DWord Ptr SS:[BX]       ; Format-Konvertierung
0074 90             NOP                               durch den 8087
0075 E888FF         CALL   PROC2                   ; Aufruf
```

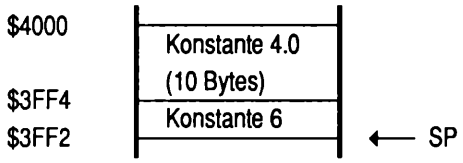
Wie gesagt - im Prinzip. *Proc2* erwartet als dritten Parameter einen Wert im Format *Single* - der Compiler hat die Konstante 2.0 aber als *Extended* gespeichert. Bei der Übergabe dieser Konstanten durch den 8087 erledigt sich die notwendige Konvertierung sozusagen von selbst: der Wert 2.0 wird (als *Extended*) mit

```
0059 2EDB2E3800     FLD    TByte Ptr CS:[0028]     ; Konstante 2.0 -> 8087
```

in das TOS geladen und innerhalb von *Proc2* mit

```
0007 D95EF2        FSTP   DWord Ptr [BP-0E]      ; Parameter Z (Single)
```

im gewünschten Format (*Single*) als »Lokalkopie« gespeichert. Die Version 5.0 tut sich da wesentlich schwerer, weil die Konvertierung hier nicht als Teil des Aufrufs ausgeführt werden kann. Der Stack vor der Ausführung des FLD-Befehls:



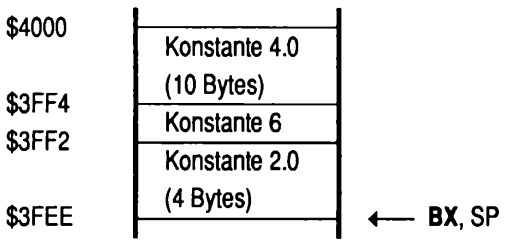
Da ein Befehl wie »PUSH TOS auf den 8086-Stack« nicht definiert ist und das BP-Register für die lokalen Variablen der aufrufenden Routine gebraucht wird, geschieht das Reservieren von Platz für den Parameter mit

```
006A 83EC04        SUB    SP,+04                 ; 4 Bytes Platz auf
006D 89E3          MOV    BX,SP                    dem Stack, BX als Zeiger
```

»zu Fuß«. BX wird als Zeiger mißbraucht - was bei der Speicherung des konvertierten Wertes mit

```
0070 36D91F        FSTP   DWord Ptr SS:[BX]      ; Format-Konvertierung
```

zusätzlich noch das Präfix `SS:` erforderlich macht. Der Stack nach diesen drei Befehlen:



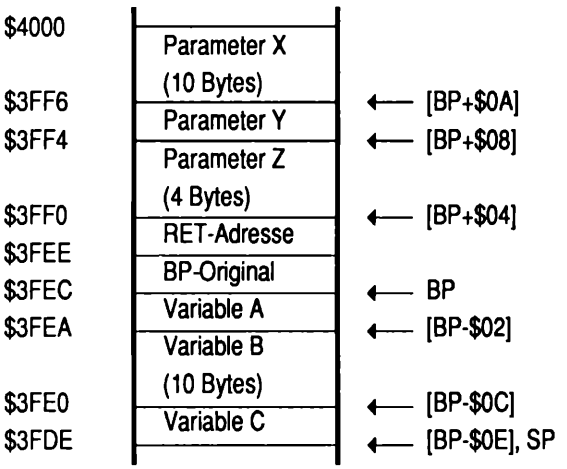
So eine Aktion vergrößert nicht nur den Code des Programms, sondern kostet vor allen Dingen Zeit - das Schlimmste, was man einem 5.0-Programm deshalb antun kann, sind implizite Format-Konvertierungen bei der Übergabe von Fließkomma-Parametern*).

Zurück zu unserer Beispiel-Prozedur, die von der Version 5.0 folgendermaßen übersetzt wird:

*) Das gilt allerdings *nicht* für Aufrufe von Bibliotheksroutinen - siehe »Geschwindigkeit und Präzision«.

```
FLOATPROC:PROC2:   *** nach Ausführung, WAIT-Befehle herausgekürzt
** begin
0000 55            PUSH   BP
0001 89E5          MOV    BP,SP
0003 83EC0E        SUB    SP,+0E                  ; 14 Bytes lokale Variablen
** B := X + Z;
0007 DB6E0A        FLD    TByte Ptr [BP+0A]       ; Parameter X -> TOS
000B D84604        FADD   DWord Ptr [BP+04]       ; Parameter Z dazu
000F DB7EF4        FSTP   TByte Ptr [BP-0C]       ; -> Variable B
** A := Y;
0014 8B4608        MOV    AX,[BP+08]              ; Parameter Y -> AX
0017 8946FE        MOV    [BP-02],AX              ; -> Variable A
** C := A;
001A 8B46FE        MOV    AX,[BP-02]              ; Variable A -> AX
001D 8946F2        MOV    [BP-0E],AX              ; -> Variable C
** end;
0020 89EC          MOV    SP,BP                   ; lokale Variablen vom Stack
0022 5D            POP    BP
0023 C21000        RET    0010                    ; Parameter X,Y,Z vom Stack
```

Und der Stack innerhalb von *Proc2* (der sich - wie gesagt - in seiner Struktur nicht von dem einer »normalen« Routine unterscheidet):



Funktionen mit Fließkomma-Ergebnistypen...

teilen den lokalen Speicherplatz in derselben Reihenfolge wie andere Funktionen auf. Der Compiler reserviert

- zuerst Platz für die Zwischenspeicherung des Funktionsergebnisses,

- danach für »Lokalkopien« übergebener Wertparameter*),
- und schließlich für direkt deklarierte lokale Variablen.

Die Funktion *Func2* des folgenden Programms soll der Prozedur *Proc2* des vorangegangenen Beispiels so weit wie möglich ähneln:

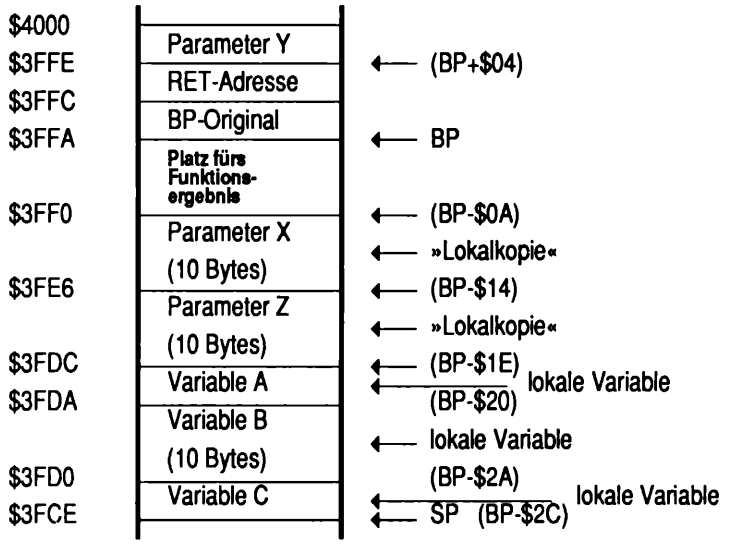
```
program FloatFunc2;
{$N+}
var
   A: Extended;

function Func2(X: Extended; Y: Integer; Z: Extended): Extended;
var
   A: Integer;
   B: Extended;
   C: Integer;
begin
  B := X + Z;
  A := Y;
  C := A;
  Func2 := B;
end;

begin
  A := Func2(4.0,6,2.0);
end.
```

Der Aufruf von *Func2* im Hauptprogramm unterscheidet sich lediglich durch die darauffolgende Zuweisung des Funktionsergebnisses:

```
   ....
** A := Func2(4.0,6,2.0);
0060 2EDB2E3F00     FLD    TByte Ptr CS:[003F]   ; Konstante 4.0 -> 8087
0065 B80600         MOV    AX,0006
0068 50             PUSH   AX                    ; Konstante 6 -> 8086-Stack
006A 2EDB2E4900     FLD    TByte Ptr CS:[0049]   ; Konstante 2.0 -> 8087
006F E88EFF         CALL   FUNC2                 ; Aufruf
0073 DB3E0000       FSTP   TByte Ptr [A]         ; Funktionsergebnis -> A
   ....
```

Innerhalb von *Func2* werden zusätzlich 10 Bytes zur Zwischenspeicherung des Funktionsergebnisses reserviert. Dementsprechend verschiebt sich die Adressierung der restlichen lokalen Variablen:

*) ... womit auch Strings gemeint sind, deren Kombination mit Fließkommawerten in diesem Buch ansonsten völlig unter den Tisch gefallen ist.



Das Listing von *Func2* beschränkt sich auf die interessanten Teile, d.h. den Start- und Endcode sowie die Zuweisung und Übergabe des Funktionsergebnisses.

```
FLOATFUNC:FUNC2:      * WAIT-Befehle herausgekürzt *
** begin
0000 55             PUSH   BP
0001 89E5           MOV    BP,SP
0003 83EC2C         SUB    SP,+2C
0007 DB7EE2         FSTP   TByte Ptr [BP-1E]     ; Parameter X
000B DB7EEC         FSTP   TByte Ptr [BP-14]     ; Parameter Z
** B := X + Z;
** A := Y;
** C := A;
    .....
** Func2 := B;
002C DB6ED6         FLD    TByte Ptr [BP-2A]     ; Variable B -> TOS
0030 DB7EF6         FSTP   TByte Ptr [BP-0A]     ; -> Funktionsergebnis
** end;
0035 DB6EF6         FLD    TByte Ptr [BP-0A]     ; Funktionsergebnis -> TOS
0039 89EC           MOV    SP,BP                 ; Lokales vom Stack
003B 5D             POP    BP
003C C20200         RET    0002                  ; Parameter Y vom Stack
```

Wie bei einer »normalen« Funktion läßt der Compiler das Funktionsergebnis immer zwischenspeichern und erst unmittelbar vor dem Rücksprung neu laden - auch dann, wenn die Routine wie hier mit einer Zuweisung endet. Das Ergebnis von *Func2* befindet sich beim Rücksprung zum aufrufenden Programm im TOS des 8087.

Version 5.0

Abgesehen von den bereits bei Prozeduren besprochenen Unterschieden gibt es hier nichts weiter zu vermelden: Die Reihenfolge beim Reservieren von Speicherplatz ist dieselbe, die Übergabe des Funktionsergebnisses findet ebenfalls über das TOS des 8087 statt. Da die Parameter durch den 8086-Stack übergeben werden und »Lokalkopien« damit entfallen, liegen die lokalen Variablen direkt unterhalb des Funktionsergebnisses auf dem Stack: *A* wird innerhalb von *Func2* also mit `[BP-$0C]` adressiert, *B* mit `[BP-$16]`...

Variablenparameter...

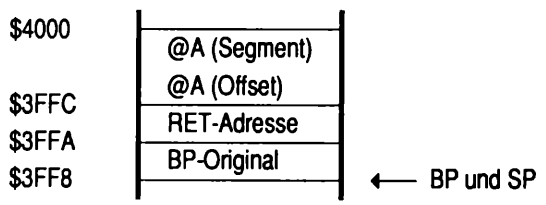
werden auch dann als ganz normale Zeiger übergeben, wenn es sich dabei um Werte im Format *Single..Extended* handelt. Ein Beispiel dazu:

```
program FloatVarParm;
{$N+}
const
  A: Extended = 4;

procedure IncFloat(var R: Extended);
begin
  R := R + 1;
end;

begin
  IncFloat(A);
end.
```

Vor dem Aufruf von *IncFloat* wird die Adresse von *A* auf den Stack gebracht:

```
    ....
** IncFloat(A);
0037 BF0000        MOV    DI,0000          ; Adresse von A auf
003A 1E            PUSH   DS                 den Stack
003B 57            PUSH   DI
003C E8CBFF        CALL   INCFLOAT         ; Aufruf
    ....
```

Die Belegung des Stacks innerhalb von *IncFloat* unterscheidet sich in keinem Punkt von einer »normalen« Prozedur, die mit einem **var**-Parameter arbeitet.



Das Ergebnis der Übersetzung von *IncFloat* durch den Compiler ist erfreulicherweise recht kurz:

```
0000  00 00 00 00 00 00 00 80 FF 3F  +0.1E+1    ; Konstante 1.0
FLOATVARPARM:INCFLOAT:  * WAIT-Befehle herausgekürzt *
** begin
000A 55            PUSH   BP
000B 89E5          MOV    BP,SP
** R := R + 1;
000E 2EDB2E0000    FLD    TByte Ptr CS:[0000]   ; Konstante 1.0 -> 8087
0013 C47E04        LES    DI,[BP+04]            ; Adresse von A -> ES:DI
0017 26DB2D        FLD    TByte Ptr ES:[DI]     ; Wert von A -> 8087
001B DEC1          FADDP  ST(1),ST              ; Addition
001D C47E04        LES    DI,[BP+04]            ; nochmal die Adresse von A
0021 26DB3D        FSTP   TByte Ptr ES:[DI]     ; Speicherung
** end;
0025 89EC          MOV    SP,BP                 ; .. keine Änderung
0027 5D            POP    BP
0028 C20400        RET    0004                  ; Zeiger vom Stack
```

Version 5.0

Wie erfreulich: Da die Variable *A* als typisierte Konstante deklariert ist und deshalb vor den typisierten Konstanten von *System* im Datensegment gespeichert wird, sind beide Versionen des Programms bis aufs letzte Bit gleich (wenn man von den zusätzlichen PUSH- und POP-Befehlen für das BP-Register im Hauptprogramm absieht).

Real-Variablen - ein schreckliches Thema

Wenn dieses Buch vor einem Jahr erschienen wäre, hätte das Kapitel 2 wohl unter umgekehrtem Vorzeichen gestanden: Einer breit angelegten Besprechung der Fließkomma-Routinen und des Typs *Real* wäre ein kurzer (und sehnsüchtiger) Ausblick auf die Möglichkeiten des 8087 gefolgt. Nun, die Zeiten ändern sich: Die 4 MHz-Version des 8087 kostet noch einen guten Hunderter, der 80287 ist je nach Grenzfrequenz für Beträge zwischen 300 und 700 Mark zu haben - und wem die Rolex mit Brillianten schon immer etwas zu schnöde war, der schielt nun auf den 80387.

Leser, die (noch) keinen Coprozessor ihr eigen nennen oder (aus welchen Gründen auch immer) auf seinen Einsatz verzichten müssen, mögen mir den Unterton auf den folgenden Seiten verzeihen - Tatsache ist, daß das Format *Real* ein echter Notbehelf war und bleibt. *Real*-Zahlen sind zu nichts in der Welt (außer der CP/M-

Version von Turbo 3.0) kompatibel, für die meisten Berechnungen zu ungenau und für den Rest zu langsam*).

Zurück auf den Teppich. Der Compiler verwendet die Register AX/BX/DX für den ersten *Real*-Wert einer Operation - wenn noch ein zweiter Wert dieses Formats beteiligt ist, wird er in den Registern CX/DI/SI gespeichert. (Diese Konvention ist dieselbe wie bei der Version 3.0).

Real-Wertparameter werden wie andere einfache Variablen vor einem Aufruf auf den Stack gebracht (obwohl sie mehr als 4 Bytes umfassen), Funktionsergebnisse läßt der Compiler direkt in den Registern AX/BX/DX zurückliefern. Die Übergabe von *Real*-Variablenparametern erfolgt (wie bei allen anderen Datentypen) über Zeiger.

Untypisierte *Real*-Konstanten speichert die Version 4.0 *direkt* in der Zielvariablen bzw. in den Registern des Prozessors (und nicht vor der jeweiligen Routine im Codesegment).

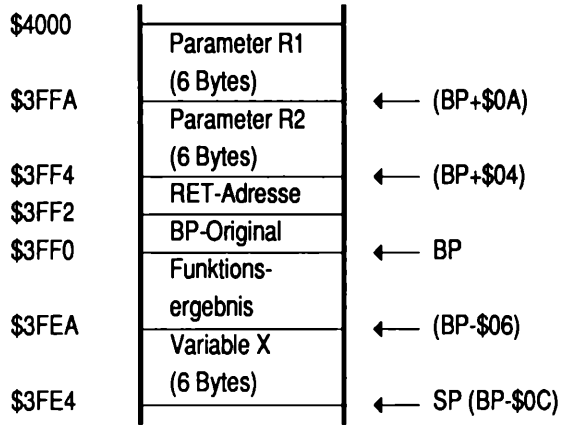
Ein vollständiges Beispiel

Das folgende Programm enthält sämtliche Elemente, die auch bereits in anderem Zusammenhang besprochen wurden: untypisierte Konstanten, lokale Variablen und einen Funktionsaufruf mit mehreren Parametern. Allzulang ist es trotzdem nicht:

```
program RealFuncDemo;

{$N-}     { ** ! ** }

var
  R: Real;

function RealFunc(R1,R2: Real): Real;
var
  X: Real;
begin
  X:= R1 + R2;
  RealFunc := X;
end;

begin
  R := 2.0;
  R:= RealFunc(R,17.0);
end.
```

*) Ganz recht: der Autor hat vor mittlerweile rund zwei Jahren einem unverschämt günstigen Angebot für einen 80287 (»aus privater Hand vom Erstbesitzer, leicht gebraucht«) nicht widerstehen können und freut sich immer noch diebisch darüber.

Die Aufteilung des Stacks zeigt keinerlei Besonderheiten - wir brauchen dieses Diagramm auch hauptsächlich für den folgenden Vergleich mit der Version 3.0.



Das Ergebnis der Übersetzung durch den Compiler:

```
REALFUNCDEMO:REALFUNC:
** begin
0000 55            PUSH   BP
0001 89E5          MOV    BP,SP
0003 83EC0C        SUB    SP,+0C              ; 6 Bytes X, 6 Bytes Ergebnis
** X:= R1 + R2;
0006 8B460A        MOV    AX,[BP+0A]
0009 8B5E0C        MOV    BX,[BP+0C]          ; Parameter R1 -> AX/BX/DX
000C 8B560E        MOV    DX,[BP+0E]
000F 8B4E04        MOV    CX,[BP+04]
0012 8B7606        MOV    SI,[BP+06]          ; Parameter R2 -> CX/SI/DI
0015 8B7E08        MOV    DI,[BP+08]
0018 9A35056B3A    CALL   SYSTEM:FADD         ; Addition, liefert das
001D 8946F4        MOV    [BP-0C],AX            Ergebnis in AX/BX/DX
0020 895EF6        MOV    [BP-0A],BX          ; -> Speicherung in X
0023 8956F8        MOV    [BP-08],DX
```

Die Routine *FAdd* (»Float Add«) in *System* erwartet den ersten Operanden in den Registern AX/BX/DX, den zweiten Operanden in den Registern CX/SI/DI und liefert ihr Ergebnis in AX/BX/DX zurück, das hier in der lokalen Variablen *X* gespeichert wird.

```
** RealFunc := X;
0026 8B46F4        MOV    AX,[BP-0C]
0029 8B5EF6        MOV    BX,[BP-0A]          ; Variable X -> AX/BX/DX
002C 8B56F8        MOV    DX,[BP-08]
002F 8946FA        MOV    [BP-06],AX
0032 895EFC        MOV    [BP-04],BX          ; -> Funktionsergebnis
0035 8956FE        MOV    [BP-02],DX
```

Das Grundschema jeder Zuweisung, an der zwei *Real*-Variablen beteiligt sind: Der Wert wird in AX/BX/DX geladen und von dort aus in die Zielvariable geschrieben.

```
** end;
0038 8B46FA        MOV    AX,[BP-06]          ; Ergebnis -> AX/BX/DX
003B 8B5EFC        MOV    BX,[BP-04]
003E 8B56FE        MOV    DX,[BP-02]
0041 89EC          MOV    SP,BP               ; Lokales vom Stack
0043 5D            POP    BP
0044 C20C00        RET    000C                ; Parameter vom Stack
```

RealFunc liefert das Funktionsergebnis in AX/BX/DX zurück, die beiden übergebenen Werte (*R1* und *R2* mit jeweils 6 Bytes) werden vom Stack entfernt.

```
REALFUNCDEMO:@:
** begin
0047 9A00006B3A    CALL   SYSTEM:@            ; Initialisierung
004C 89E5          MOV    BP,SP
** R := 2.0;
004E C70600008200  MOV    Word Ptr [R],0082   ; direkte Zuweisung:
0054 C70602000000  MOV    Word Ptr [R+02],0000  82 00 00 00 00 00 = .2E+1
005A C70604000000  MOV    Word Ptr [R+04],0000
```

Das Grundschema jeder Zuweisung einer untypisierten Konstanten an eine *Real*-Variable: der Compiler setzt die einzelnen Werte direkt.

```
** R:= RealFunc(R,17.0);
0060 FF360400      PUSH   [R+04]
0064 FF360200      PUSH   [R+02]              ; Wert von R auf den Stack
0068 FF360000      PUSH   [R]
006C B88500        MOV    AX,0085             ; 85 00 00 00 00 08 = .17E+2
006F 31DB          XOR    BX,BX                 in AX/BX/DX
0071 BA0008        MOV    DX,0800
0074 52            PUSH   DX
0075 53            PUSH   BX                  ; und auf den Stack
0076 50            PUSH   AX
0077 E886FF        CALL   REALFUNC            ; Aufruf, zurück mit AX..DX
007A A30000        MOV    [R],AX
007D 891E0200      MOV    [R+02],BX           ; Zuweisung an R
0081 89160400      MOV    [R+04],DX
```

Der Aufruf der Funktion *RealFunc* mit einer Variablen und einer untypisierten Konstanten: Für die Übergabe der Konstanten verwendet der Compiler auch hier das »primäre *Real*-Register« (AX/BX/DX). *RealFunc* liefert das Ergebnis auch in diesem »Register« zurück.

```
** end.
0085 89EC          MOV    SP,BP
0087 31C0          XOR    AX,AX               ; Exitcode 0
0089 9AF3016B3A    CALL   SYSTEM:HALT         ; Programmende
```

Version 5.0

Bis auf die PUSH- und POP-Befehle für das BP-Register im Hauptprogramm gibt es hier nicht den geringsten Unterschied.

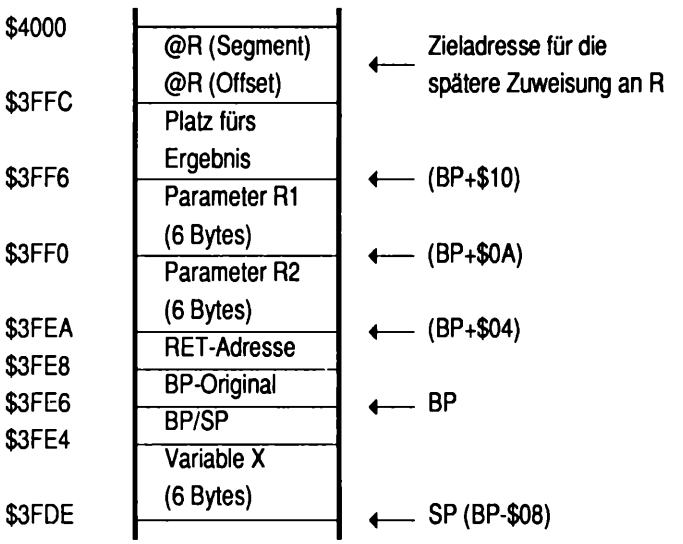
Version 3.0

Die Version 3.0 verwendet ebenfalls die Register AX/BX/DX für den ersten und die Register CX/SI/DI für den zweiten Operanden einer *Real*-Berechnung. Vom »sekundären *Real*-Register« bekommt man allerdings nur in der Laufzeitbibliothek etwas zu sehen, weil *Real*-Werte wie Strings zu hundert Prozent über den Stack transportiert werden.

Zwischen den beiden Versionen des Compilers bestehen hauptsächlich drei Unterschiede im Bezug auf *Real*-Berechnungen:

- *Real*-Konstanten (egal ob typisiert oder untypisiert) werden wie Strings inmitten des Programmcodes gespeichert.
- Die Übergabe aller Parameter und Ergebnisse läuft ausschließlich über den Stack. Die Routinen die Laufzeitbibliothek erwarten die beiden Operanden ebenfalls auf dem Stack (d.h. nicht in Registern des Prozessors) und liefern ihr Ergebnis auch wieder auf dem Stack zurück.
- Wie bei allen 3.0-Funktionen wird der Platz zur Zwischenspeicherung des Ergebnisses von der *aufrufenden* Routine zur Verfügung gestellt. Da *Real*-Funktionen ihr Ergebnis auf dem Stack zurückliefern (sozusagen als strukturierter Wert), wird der Platz nicht innerhalb der Funktion, sondern erst durch die aufrufende Routine wieder freigegeben (vgl. »Funktionen« in Kapitel 1).

Die Aufteilung des Stacks innerhalb der Funktion *RealFunc*:



Die Übersetzung von *RealFuncDemo* durch die Compilerversion 3.01A (mit abgeschalteter Stackprüfung):

```
2D9C E93B00        JMP    2DDA              ; Sprung zum Hauptprogramm

REALFUNCDEMO:REALFUNC:
** begin
2D9F 55            PUSH   BP
2DA0 8BEC          MOV    BP,SP
2DA2 55            PUSH   BP
2DA3 E90000        JMP    2DA6
2DA6 83EC06        SUB    SP,+06            ; 6 Bytes Platz für X
** X:= R1 + R2;
2DA9 8D7EF8        LEA    DI,[BP-08]        ; Adresse der Variablen
2DAC 16            PUSH   SS                  X (SS:BP-$08) als Ziel
2DAD 57            PUSH   DI                  auf den Stack
2DAE 8D7E0A        LEA    DI,[BP+0A]        ; Adresse des Parameters
2DB1 16            PUSH   SS                  R1 (SS:BP+$0A) -> Stack/DI
2DB2 E8BEEB        CALL   1973              ; RealToStack
2DB5 8D7E04        LEA    DI,[BP+04]        ; dasselbe mit dem
2DB8 16            PUSH   SS                  Parameter R2
2DB9 E8B7EB        CALL   1973              ; RealToStack
2DBC E8E7EB        CALL   19A6              ; FADD (Addition)
2DBF E8D1EB        CALL   1993              ; StackToRealVar
```

Die Routine *RealToStack* erwartet den Segment-Anteil der zu lesenden Variablen auf dem Stack und den Offset-Anteil im Register DI. Sie holt den Segment-Anteil wieder vom Stack und legt den Wert der Variablen darauf, erniedrigt SP also um 4 Bytes. *FAdd* holt beide *Real*-Werte vom Stack, addiert sie und legt das Ergebnis wieder auf den Stack. *StackToRealVar* erwartet eine Zieladresse (hier: die Adresse von *X*, die am Anfang der Operation auf den Stack gelegt wurde) und einen *Real*-Wert, holt beide Parameter vom Stack und führt die Speicherung aus.

```
** RealFunc := X;
2DC2 8D7E10        LEA    DI,[BP+10]        ; Adresse des Funktions-
2DC5 16            PUSH   SS                  ergebnisses als Ziel
2DC6 57            PUSH   DI                  auf den Stack
2DC7 8D7EF8        LEA    DI,[BP-08]        ; Adresse von X -> Stack/DI
2DCA 16            PUSH   SS
2DCB E8A5EB        CALL   1973              ; RealToStack - lädt X
2DCE E8C2EB        CALL   1993              ; StackToRealVar
```

Das Grundschema einer Zuweisung zwischen zwei *Real*-Variablen ist etwas übersichtlicher als eine Addition: Zuerst wird die Adresse der Zielvariablen auf den Stack gebracht, dann lädt *RealToStack* den neuen Wert dazu, der schließlich über *StackToRealVar* in die Zielvariable geschrieben wird.

```
** end;
2DD1 E90000        JMP    2DD4
2DD4 8BE5          MOV    SP,BP                ; Lokales vom Stack
2DD6 5D            POP    BP
2DD7 C20C00        RET    000C                 ; R1 und R2 vom Stack
```

Das Funktionsergebnis bleibt hier in dem Bereich stehen, den die aufrufende Routine dafür reserviert hat (d.h. direkt oberhalb der übergebenen Parameter). Die Parameter selbst werden beim Rücksprung auf normale Weise entfernt.

```
REALFUNCDEMO:@
** begin
** R := 2.0;
2DDA BF6002        MOV    DI,0260              ; Adresse von R
2DDD 1E            PUSH   DS                     auf den Stack
2DDE 57            PUSH   DI
2DDF E8A0EB        CALL   1982                 ; CSegRealToStack
2DE2 82 00 00 00 00 00            ; 0.2E+1
2DE8 E8A8EB        CALL   1993                 ; StackToRealVar
```

Das Grundschema der Zuweisung einer untypisierten *Real*-Konstanten an eine Variable: Zuerst wird die Zieladresse auf den Stack gebracht (und beim Abschluß der Operation von *StackToRealVar* wieder gelesen). Die Routine *CSegRealToStack* verwendet die RET-Adresse des aufrufenden Programms als Zeiger auf die Konstante, bringt den Wert auf den Stack und *erhöht* die RET-Adresse entsprechend - danach geht es also mit dem Aufruf von *StackToRealVar* weiter.

```
** R := RealFunc(R, 17.0);
2DEB BF6002        MOV    DI,0260              ; Adresse von R als
2DEE 1E            PUSH   DS                     Ziel auf den Stack
2DEF 57            PUSH   DI
2DF0 83EC06        SUB    SP,+06               ; 6 Bytes fürs Ergebnis
2DF3 BF6002        MOV    DI,0260              ; Adresse von R -> Stack/DI
2DF6 1E            PUSH   DS
2DF7 E879EB        CALL   1973                 ; RealToStack
2DFA E885EB        CALL   1982                 ; CSegRealToStack
2DFD 85 00 00 00 00 08            ; 0.17E+2
2E03 E899FF        CALL   REALFUNC
2E06 E88AEB        CALL   1993                 ; StackToRealVar
```

Das Schema einer Zuweisung ist auch bei Funktionsaufrufen dasselbe: Zuerst wird die Adresse der Zielvariablen auf den Stack gebracht. Die Funktion liefert ihr Ergebnis auf dem Stack, *StackToRealVar* führt die eigentliche Zuweisung aus (und erhöht SP wieder entsprechend).

Die Übergabe der beiden Parameter verläuft genauso wie die Addition in *RealFunc*: *RealToStack* bringt den Wert der Variablen *R* auf den Stack (wobei der Offset-Anteil der Variablenadresse via DI direkt übergeben wird); *CSegRealToStack* liest die im Codesegment folgende Konstante 17.0, legt sie auf den Stack und er-

höht die RET-Adresse um sechs, damit der Prozessor nicht beim Rücksprung in der Konstanten selbst landet.

```
** end.
2E09 E90000        JMP    2E0C
2E0C 33C0          XOR    AX,AX              ; Exitcode 0
2E0E E878DE        CALL   0C89               ; HALT
```

Real-Variablen im Modus {$N+}

Ein Zitat aus dem 4.0-Referenzhandbuch, Kapitel 24: »Wenn ein 8087 auf Variablen des Typs *Real* angewendet wird, steigt die Ausführungsgeschwindigkeit des Programms deutlich - allerdings nicht so stark, wie man es eigentlich erwarten sollte.«

Frei nach Radio Eriwan: Im Prinzip könnte sie steigen, wenn der Compiler nicht nachhaltig dafür sorgen würde, daß sie es nicht tut. Tatsächlich sinkt sie - und zwar kräftig*).

Auf einem AT mit 12 MHz (bei dem der 80287 wie üblich mit 2/3 der Taktfrequenz läuft), sind 100 000 Durchläufe des Hauptprogramms von *RealFuncDemo* (siehe letzter Abschnitt bzw. unten) in rund 6 Sekunden erledigt, wenn der Modus {$N-} gesetzt ist - bei aktiviertem Coprozessor wird die Ausführung so »beschleunigt«, daß sich der Computer erst nach 21 Sekunden wieder meldet.

Sehen wir uns kurz an, warum dem so ist. Zuerst eine Wiederholung von *RealFuncDemo*, wobei dieses Programm lediglich um den Compilerbefehl {$N+} erweitert wurde:

```
program RealFuncDemo;
{$N+}
var
  R: Real;

function RealFunc(R1,R2: Real): Real;
var X: Real;
begin
  X:= R1 + R2;
  RealFunc := X;
end;

begin
  R := 2.0;
  R:= RealFunc(R,17.0);
end.
```

*) Wenn man nicht jeden Mist selber nachprüft - ist in der Version 5.0 zwar nicht mehr ganz so schlimm, aber trotzdem korrigiert.

Der Beginn des Hauptprogramms:

```
0061  00 00 00 00 00 00 00 80 00 40  +0.2E+1
006B  00 00 00 00 00 00 00 88 03 40  +0.17E+2
REALFUNCDEMO:@:
** begin
0075 9A00006F3A     CALL   SYSTEM:@              ; Initialisierung
007A 9AC3056F3A     CALL   SYSTEM:INIT87         ; Init 8087
007F 89E5           MOV    BP,SP
** R := 2.0;
0081 9B             WAIT
0082 2EDB2E6100     FLD    TByte Ptr CS:[0061]   ; Extended-Konstante 2.0
0087 9AE4066F3A     CALL   SYSTEM:06E4           ; Extended -> Real
008C A30000         MOV    [R],AX
008F 891E0200       MOV    [R+02],BX             ; Zuweisung an R
0093 89160400       MOV    [R+04],DX
```

Hier haben wir den ersten Grund für den inversen Geschwindigkeitszuwachs: Der Compiler unterscheidet bei untypisierten Konstanten offensichtlich nur zwischen »Integer« und »Realzahl«. Weil der Modus {$N+} gesetzt ist, speichert er den Wert 2.0 im Format *Extended* und muß ihn für die Zuweisung an die *Real*-Variable *R* erst umrechnen. (Wie im Abschnitt »Typisierte und untypisierte Konstanten« in Kapitel 1 erläutert, hätte die Angabe »2« anstelle von »2.0« dasselbe Ergebnis).

Der technische Hintergrund: Der Compiler hat im Modus {$N+} nur einen einzigen Generatorzweig für Zuweisungen an *Real*-Variablen. Die rechte Seite der Zuweisung wird immer im Format *Extended* evaluiert und vor der Zuweisung in das Format *Real* umgewandelt - unabhängig davon, ob diese rechte Seite aus einer einzelnen Konstanten, einem Funktionsaufruf oder einer zusammengesetzten Berechnung besteht.

Um mit diesem einen Generatorzweig auszukommen, muß der Compiler dafür sorgen, daß die rechte Seite einer Zuweisung immer das Format *Extended* hat - und hier haben sich die Programmierer von Borland die Sache allzu einfach gemacht. Der folgende Ausschnitt von *RealFunc* zeigt das haarsträubende Ergebnis:

```
REALFUNCDEMO:REALFUNC:
** begin
0000 55             PUSH   BP
0001 89E5           MOV    BP,SP
0003 83EC0C         SUB    SP,+0C                ; 6 Bytes X, 6 Bytes Ergebnis
** X:= R1 + R2;
0006 8B4604         MOV    AX,[BP+04]
0009 8B5E06         MOV    BX,[BP+06]            ; Parameter R1 -> AX/BX/DX
000C 8B5608         MOV    DX,[BP+08]
000F 9AAF066F3A     CALL   SYSTEM:06AF           ; Real -> Extended
0014 9AE4066F3A     CALL   SYSTEM:06E4           ; Extended -> Real
0019 52             PUSH   DX
001A 53             PUSH   BX                    ; Ergebnis auf den Stack
001B 50             PUSH   AX
```

```
001C 8B460A        MOV    AX,[BP+0A]
001F 8B5E0C        MOV    BX,[BP+0C]          ; Parameter R2 -> AX/BX/DX
0022 8B560E        MOV    DX,[BP+0E]
0025 9AAF066F3A    CALL   SYSTEM:06AF         ; Real -> Extended
002A 9AE4066F3A    CALL   SYSTEM:06E4         ; Extended -> Real
002F 59            POP    CX
0030 5E            POP    SI                  ; Ergebnis der Umwandlung
0031 5F            POP    DI                    R1 ins zweite Real-Register
0032 9A35056F3A    CALL   SYSTEM:FADD         ; Addition R1 / R2
0037 8946F4        MOV    [BP-0C],AX
003A 895EF6        MOV    [BP-0A],BX          ; Ergebnis von FADD -> X
     .....
```

So kann eine harmlose Addition also auch aussehen - nämlich dann, *wenn beide Operanden bereits das Format* Real *haben.* Der Compiler besteht darauf, vor den Aufruf von *FAdd* eine Konvertierung von *Extended* nach *Real* zu setzen und muß deshalb vorher dafür sorgen, daß die zu konvertierenden Werte auch das Format *Extended* haben.

Verzichten wir auf eine weitere Klassifizierung dieser Methodik. Wenn ein Programm im Modus {$N+} ohne *Real*-Variablen absolut nicht auskommt (z.B. weil eine Datei der Version 3.0 gelesen werden muß), dann sollte dieser Variablentyp nur an den Stellen verwendet werden, wo es wirklich nötig ist. Oder auch: Lieber einen Wert dreimal über *Single..Extended* hin- und herschaufeln als eine einzige Addition (oder Zuweisung!) zwischen *Reals*.

Version 5.0

Offensichtlich sind die Probleme der Version 4.0 mit *Real*-Variablen auch den Entwicklern von Borland aufgefallen - hier hat sich einiges getan:

❑ Der Compiler unterscheidet bei untypisierten numerischen Konstanten nicht mehr zwischen zwei, sondern zwischen drei Klassen: »Integer«, »IEEE-Fließkommazahl« und »Real«. Die Zuweisung einer Konstanten an eine Variable des Typs *Real* fällt deshalb auch im Modus {$N+} so aus, wie man es eigentlich erwarten sollte:

```
   ....
** R:= 2.0;
0061 C7063C008200   MOV    Word Ptr [R],0082      ; direkte Zuweisung
0067 C7063E000000   MOV    Word Ptr [R+02],0000     an R - ohne EXTENDED-
006D C70640000000   MOV    Word Ptr [R+04],0000     Konstante & Konvertierung
```

❑ Der an sich notwendige Generatorzweig für Rechenoperationen mit *Real*-Variablen wurde durch eine *Kürzung* realisiert: Der Compiler wandelt nach wie vor jeden *Real*-Wert als erstes in *Extended* um, läßt die Berechnung dann aber nicht über eine Bibliotheksroutine ausführen, sondern benutzt den 8087 direkt:

```
   ....
** X:= R1+R2;        *** wie üblich *nach* Ausführung des Programms
0006 8B4604          MOV    AX,[BP+04]
0009 8B5E06          MOV    BX,[BP+06]            ; Parameter R1 -> AX/BX/DX
000C 8B5608          MOV    DX,[BP+08]
000F 9AFA036C48      CALL   SYSTEM:03FA           ; Real -> Extended -> TOS
0014 8B460A          MOV    AX,[BP+0A]
0017 8B5E0C          MOV    BX,[BP+0C]            ; Parameter R2 -> AX/BX/DX
001A 8B560E          MOV    DX,[BP+0E]
001D 9AFA036C48      CALL   SYSTEM:03FA           ; Real -> Extended -> TOS
0022 9B              WAIT
0023 DEC1            FADDP  ST(1),ST              ; Addition R1 / TOS
0025 9A24046C48      CALL   SYSTEM:0424           ; TOS -> AX/BX/DX (Real)
002A 8946F4          MOV    [BP-0C],AX
002D 895EF6          MOV    [BP-0A],BX            ; Speicherung in X
0030 8956F8          MOV    [BP-08],DX
   .....
```

Das Ergebnis dieser beiden Optimierungen kann sich sehen lassen: 100000 Durchläufe des Hauptprogramms von *RealFuncDemo* werden im Modus {$N+} nicht mehr in 21, sondern in knapp 10 Sekunden erledigt.

Geschwindigkeit und Präzision

Im Abschnitt über das Kontrollwort-Register des 8087 wurde beiläufig erwähnt, daß sich die Rechengenauigkeit des Coprozessors über die beiden Bits »Precision Control« auf *Single*, *Double* oder *Extended* einstellen läßt. Nach dem Befehl FINIT arbeitet der 8087 mit der Genauigkeit *Extended* (PC = 03), der von INIT87 auch dann nicht verändert wird, wenn das gesamte Programm mit einer Umdeklaration von *Extended* in *Single* arbeitet. Ob sich da noch etwas an Geschwindigkeit herausholen läßt? Um das Ergebnis vorwegzunehmen: Nein - eher das Gegenteil. Nach einer Umdeklaration von *Extended* in *Single* mit

```
type Extended = Single;
```

zu Beginn eines Programms muß der 8087 zwar nur noch 4 anstelle von 10 Bytes für jeden Operanden bearbeiten (also weniger als die Hälfte) - in puncto Geschwindigkeit bringt das aber erstaunlich wenig. Eine Schleife wie

```
for x:= 1 to 100 do
  for y := 1 to 1000 do
    Z: = (A * B * C) / C + B;
```

wird nach der Deklaration von *Z*, *A*, *B* und *C* als *Extended* in 13.8 Sekunden ausgeführt, bei der Deklaration dieser Variablen als *Double* sind es 12.1 und nach einer Deklaration als *Single* immerhin noch 10.1 Sekunden, die ein AT mit 12 MHz dafür braucht.

Wer nun einwendet, daß der 8087 in allen drei Fällen nach wie vor intern mit der Genauigkeit von *Extended* arbeitet, hat vollkommen recht. Umso verblüffender sind die Ergebnisse des folgenden Programms*):

```
program Precision8087;  { PREC8087.PAS }
{$N+}
(* type Extended = .... *)

const
  MAX = 20;  { "Wurzeltiefe" }
  A: Extended = 1.23456789012345678901234567890;

var v,x,y,z: Integer;
    SysTicks: LongInt absolute $40:$6C;   { zur Zeitmessung }
    TickStart, TickEnd: LongInt;

    CWord: Word;  { zum Setzen des Kontrollwort-Registers }
    B: Extended;  { für Zwischenergebnisse der Berechnung }

procedure SetPrecision(p: Byte);
{ Setzt die Rechengenauigkeit des 8087. Erlaubte Werte für p: 1..3 }
begin
  CWord:= p shl 8 + $72;
  inline($9B/$D9/$2E/CWord/$9B);  { FWAIT / FLDCW[CWord] / FWAIT }
end;

begin
  for z:= 1 to 3 do   { von Single.. Extended }
  begin
    SetPrecision(z);
    Write(z,' ');

 { 1. Messung der Genauigkeit }
    b:= a;
    for v:= 1 to MAX do b:= Sqrt(b);     { die rund einmillionste Wurzel }
    for v:= 1 to MAX do b:= b * b;       { und wieder "hochquadrieren" }
    Write(b,'  ');

 { 2. Messung der Geschwindigkeit }
    b:= a;
    TickStart := SysTicks;           { Start-"Uhrzeit" }
    for x:= 1 to 10 do
      for y := 1 to 1000 do
        for v:= 1 to MAX do
```

*) ... bei dem es sich nicht um einen breit angelegten Weg aufs Glatteis handelt. Der 8087 scheint bei anderen Formaten und Genauigkeiten als *Extended* intern konvertieren zu müssen, wobei der größte Teil des Zeitgewinns durch die kürzeren Lade- und Speicheroperationen wieder verloren geht.

```
        begin
          b := Sqrt(b);          { 200 000 Wurzeln und Quadrate }
          b := b * b;
        end;
    TickEnd := SysTicks;                { End-"Uhrzeit" }
    Writeln(((TickEnd-TickStart) / 18.23):0:2, ' Sekunden.');
   end;
  end.
```

(Die Zeitmessung greift direkt auf einen Zähler zu, der vom BIOS beim Start des Computers auf 0 gesetzt und durch jeden »Clock Tick« um eins erhöht wird. Da das Zeitraster bei einer knappen zwanzigstel Sekunde liegt, ist die zweite Dezimalstelle der gemessenen Werte mit Vorsicht zu genießen).

Die Ergebnisse des Programms für »*Extended* = *Extended*«:

		Version 4.0	Version 5.0
1	1.13311278820038E+0000	22.27 Sekunden	19.42 Sekunden
2	1.23456789006442E+0000	23.37 Sekunden	20.57 Sekunden
3	1.23456789012338E+0000	22.93 Sekunden	20.08 Sekunden

Für *Extended* = *Double*:

		Version 4.0	Version 5.0
1	1.13311278820038E+0000	23.04 Sekunden	20.19 Sekunden
2	1.23456789006442E+0000	24.14 Sekunden	20.28 Sekunden
3	1.23456789006442E+0000	23.70 Sekunden	20.84 Sekunden

Und schließlich für *Extended* = *Single*:

		Version 4.0	Version 5.0
1	1.13311278820038E+0000	21.23 Sekunden	18.43 Sekunden
2	1.13311278820038E+0000	21.23 Sekunden	18.43 Sekunden
3	1.13311278820038E+0000	21.01 Sekunden	18.05 Sekunden

Wie sich an den Rechenergebnissen erkennen läßt, entsprechen die Typen *Single*, *Double* und *Extended* exakt den Genauigkeitsstufen des 8087: für *Extended* ist das Ergebnis bei »PC = 3« am genauesten; wenn die Variablen als *Double* deklariert sind, ändert die Erhöhung von »PC« von 2 auf 3 nichts mehr am Ergebnis, bei *Single*-Variablen spielt der Wert von »PC« überhaupt keine Rolle mehr.

Da die zeitlichen Unterschiede bei weniger als 10 Prozent liegen, erscheint eine Wertung sowieso nicht sonderlich sinnvoll - auffällig ist aber trotzdem, daß sich der 8087 mit dem Format *Double* am schwersten tut.

Fazit: In den Fällen, wo es wirklich auf Geschwindigkeit ankommt und *Single*-Variablen zu ungenau sind, sollte nicht als »nächste Stufe« *Double*, sondern *Extended* verwendet werden.

Version 5.0

Wie die (im vorangehenden Abschnitt daneben geschriebenen) Zeitmessungen zeigen, schlägt sich zumindest die Funktion *Sqrt* in der Version 5.0 hervorragend - offensichtlich haben die Entwickler hier noch einmal recht kräftig geputzt. Über-

lassen wir das Thema »Zeitmessungen mit dem Emulator« und die dazugehörigen (vergleichsweise katastrophalen) Ergebnisse dem nächsten Abschnitt und kommen wir zu einem Punkt, der auf Seite 126 lediglich in einer Fußnote angedeutet wurde: Bibliotheksroutinen wie *Sqrt* erwarten ihre Parameter im Format *Extended* - folglich müßte der Compiler nach einer Deklaration der Programmvariablen *b* als *Single* eigentlich eine entsprechende Konvertierung voranstellen. Was dabei herauskommen sollte, demonstriert das folgende Programm:

```
program RTLCallDemo;
{$N+}

const   { Diese Deklaration muß jeweils geändert werden }
  B: Extended = 1.23456789012345678901234567890;

var x: LongInt;
    SysTicks: LongInt absolute $40:$6C;   { zur Zeitmessung }
    TickStart, TickEnd: LongInt;

function MySqrt(x: Extended): Extended;
begin
  MySqrt := Sqrt(x);
end;

begin
   TickStart := SysTicks;           { Start-"Uhrzeit" }
   x:= 0;
   while x < 200000 do
   begin
     b := MySqrt(b);        { 200 000 Wurzeln und Quadrate }
     b := b * b;
     Inc(x);
   end;
   TickEnd := SysTicks;              { End-"Uhrzeit" }
   Writeln(((TickEnd-TickStart) / 18.23):0:2, ' Sekunden.');
end.
```

Die Schleife zur Zeitmessung ist im Prinzip dieselbe wie in *Precision8087* - nur daß *Sqrt* hier durch eine eigene Routine (*MySqrt*) ersetzt wurde, die ihrerseits aus einem Aufruf von *Sqrt* besteht. Die Ergebnisse je nach Typ der Konstanten *B*:

Typ von *B*	Version 4.0	Differenz	Version 5.0	Differenz
Extended	32,20 Sekunden	9,93	26,82 Sekunden	7,4
Double	33,19 Sekunden	10,15	30,61 Sekunden	10,42
Single	30,17 Sekunden	9,16	27,81 Sekunden	9,38

Während der zusätzliche Zeitbedarf in der Version 4.0 in etwa linear ausfällt, benötigt die Version 5.0 beim Wechsel von *Extended* nach *Double* fast 30 Prozent

länger für den Aufruf - wie bereits im Abschnitt »Gemischte Parameter« erläutert, bemüht der Compiler hier den Coprozessor zur Format-Konvertierung und speichert das Ergebnis anschließend wieder auf dem 8086-Stack:

```
    ....
*** b := MySqrt(b);
0072 DD060000      FLD    QWord Ptr [B]          ; B (DOUBLE) -> TOS
0076 83EC0A        SUB    SP,+0A                 ; 10 Bytes Platz auf dem
0079 89E3          MOV    BX,SP                    Stack für EXTENDED
007C 36DB3F        FSTP   TByte Ptr SS:[BX]      ; TOS (EXTENDED) -> Stack
0081 E87CFF        CALL   MYSQRT                 ; Aufruf
0085 DD1E0000      FSTP   QWord Ptr [B]          ; Ergebnis (TOS) -> B
    ....
```

Wieso sich in den Meßergebnissen des Programms *Precision8087* kein Hinweis auf diese Problematik findet? Weil der Compiler hier zweigleisig fährt*): **Alle Bibliotheksroutinen bekommen ihre Parameter wie in der Version 4.0 durch die Register des 8087 übergeben.** Diese Behauptung gilt nicht nur für arithmetische Routinen, wie das folgende Minimalprogramm beweist:

```
var a:Extended; b: Double; c: Single; r: Real;
begin
  Writeln(a,b,c,r);
end.
```

Die Übersetzung des Aufrufs von *Writeln* durch den Compiler:

```
    ....
*** Writeln(a,b,c,r);     *** nach Ausführung, WAIT-Befehle gekürzt
000D BF5801        MOV    DI,0158
0010 1E            PUSH   DS                     ; Adresse von OUTPUT
0011 57            PUSH   DI                       auf den Stack
0013 DB2E3C00      FLD    TByte Ptr [A]          ; Variable A -> TOS
0017 B81700        MOV    AX,0017                ; Formatlänge
001A 50            PUSH   AX
001B B8FFFF        MOV    AX,FFFF                ; Nachkommastellen
001E 50            PUSH   AX
001F 9AA90D4154    CALL   SYSTEM:WRITE
0025 DD064600      FLD    QWord Ptr [B]          ; Variable B -> TOS
0029 B81700        MOV    AX,0017
    ...
0031 9AA90D4154    CALL   SYSTEM:WRITE
0037 D9064E00      FLD    DWord Ptr [C]          ; Variable C -> TOS
003B B81700        MOV    AX,0017
    ...
0043 9AA90D4154    CALL   SYSTEM:WRITE
```

*) Was vom Standpunkt der reinen Lehre nichts weiter als ein schmutziger Trick ist - aber wie man sieht, funktioniert die Sache.

```
0048 A15200        MOV    AX,[R]
004B 8B1E5400      MOV    BX,[R+02]              ; Variable R -> AX/BX/DX
004F 8B165600      MOV    DX,[R+04]
0053 9A50044154    CALL   SYSTEM:0450            ; AX/BX/DX -> TOS
0058 B81700        MOV    AX,0017
   ...
0060 9AA90D4154    CALL   SYSTEM:WRITE
0065 9A080D4154    CALL   SYSTEM:CROUT           ; CR hintendran
006A 9A0E024154    CALL   SYSTEM:IORESULT        ; Prüfung von InOutRes
   ...
```

Für Benchmark-Tester

Nachdem die technischen Möglichkeiten, die Problematik mit *Real*-Zahlen usw. nun eingehendst abgehandelt sind, sollten wir vielleicht noch einmal auf die »Geschwindigkeitssteigerung um den Faktor 10 (oder mehr)« zurückkommen. Schreiben wir also ein kurzes (und nicht hundertprozentig repräsentatives) Meßprogramm:

```
program Speed87;   { SPEED87.PAS }
{$N+}                 { ** ABSCHALTEN für Real = Real! }
type
   Real = Extended;
var
   y,z: Real;
   a,b: Integer;

  SysTicks: LongInt absolute $40:$6C;   { zur Zeitmessung }
  TickStart, TickEnd: LongInt;

begin
  Writeln('Zeitmessung läuft...');
  TickStart := SysTicks;

  y := 1.3452;
  for a:= 1 to 100 do      { 100000 Durchläufe }
   for b:= 1 to 1000 do
     z := 2.5 * y + 2.4;

  TickEnd := SysTicks;
  Writeln(((TickEnd-TickStart) / 18.23):0:2, ' Sekunden.',#7);
end.
```

Was den direkten Vergleich zwischen den Modi {$N+} und {$N-} bei »type *Real* = *Real*« betrifft, haben wir ja bereits festgestellt, daß die Ergebnisse ziemlich niederschmetternd sind. Wie sieht es mit einem Vergleich zwischen *Real* (im Modus {$N-}) und *Single* (im Modus {$N+}) aus? Das Ergebnis ist beachtlich (21 zu 7

Sekunden), aber von einem »Faktor 10« doch recht weit entfernt. Die Suche nach möglichen Gründen[1] für dieses magere Ergebnis fördert nur dann etwas zutage, wenn man Turbo Pascal mit anderen Programmen vergleicht. Äquivalente Versionen des Programms *Speed87* brauchen

- bei einer Compilierung mit der Version 3.0 (ohne Coprozessor und dem Typ *Real*): 28 Sekunden (Geschwindigkeitsunterschied: 1 zu 4);
- bei einer Compilierung mit Turbo Basic (mit Coprozessor und dem Typ *Single*): 14 Sekunden - Turbo Pascal holt also *rund das Doppelte* aus dem 8087 heraus;
- bei einer Compilierung mit Turbo Basic (ohne Coprozessor): rund 50 Sekunden (Geschwindigkeitsunterschied: 1 zu 7).

Fazit: Die Version 4.0 erzeugt zusammen mit dem 8087 ausgesprochen schnelle Programme - die *Real*-Arithmetik (ohne Coprozessor) ist aber bereits so stark optimiert, daß der Einsatz des 8087 das Programm »nur noch« um das dreifache beschleunigt.

Version 5.0

Da der Löwenanteil der Laufzeit von *Speed87* auf direkte 8087-Operationen entfällt, sind die gemessenen Zeiten in etwa dieselben - was nachzutragen bleibt, ist das Ergebnis bei Verwendung des Emulators:

Nach einer »Umdeklaration« von *Real* in *Extended* benötigt *Speed87* rund 46.2 Sekunden zur Ausführung auf einem System ohne Coprozessor (und ansonsten gleichen Bedingungen wie Taktfrequenz, Wartezyklen etc.) - also ziemlich genau die doppelte Zeit wie für ein »Real-Programm«.

Und eine Erkenntnis zum Schluß: Die Behauptung von Seite 126, daß implizite Typ-Konvertierungen von Fließkomma-Parametern so ziemlich das Schlimmste sind, was man einem 5.0-Programm antun kann, läßt sich noch steigern - nämlich dadurch, daß man diese Konvertierungen dem Emulator auflädt. Das im letzten Abschnitt vorgestellte Programm *RTLCallDemo* benötigt nach der Deklaration der Variablen *B* als *Single* auf einem System ohne Coprozessor die sagenhafte Zeit von 186.5 Sekunden[2].

---

[1] ... bei der ich sogar mit einem Oszilloskop und einem Frequenzzähler auf meinen AT losgegangen bin, weil ich es am Anfang einfach nicht glauben wollte...

[2] Bei der Übergabe eines *Extended*-Wertes als *Single*-Parameter sieht die Arbeit des Emulators fast schon hanebüchen aus: Nach dem Laden des Wertes ins (simulierte) TOS, der Umrechnung ins Format *Single* und Speicherung auf dem 8087-Stack bekommt er den soeben konvertierten Wert innerhalb der aufgerufenen Routine erneut serviert - und rechnet ihn dort erstmal wieder in *Extended* um...

Direkte Programmierung

Wie bereits zu Anfang dieses Kapitels gesagt, lastet Turbo Pascal den Coprozessor für eine Hochsprache außergewöhnlich gut aus - Überwältigendes ist deshalb auch bei einer ausgetüftelten manuellen Optimierung nicht zu erwarten*).

Der Compiler hat drei Schwächen, die alle nichts mit Turbo Pascal zu tun haben, sondern prinzipbedingt sind:

- Zusammenhänge zwischen aufeinanderfolgenden Anweisungen werden nicht berücksichtigt. Eine Befehlsfolge wie

  ```
  A := B;  C:= B;
  ```

  wird als »B laden, in A speichern«, »B laden, in C speichern« codiert. Der Programmierer weiß im Gegensatz zum Compiler, daß der Wert von B gleich noch einmal (und unverändert!) gebraucht wird, er kann deshalb eine Ladeoperation einsparen.

- Mehrere Vorkommen eines Wertes in einem Ausdruck werden separat behandelt. In einer Anweisung wie

  ```
  A : = B * B;
  ```

  macht der Compiler keinen Gebrauch von der Möglichkeit 8087-interner Registerkopien und erzeugt zweimal hintereinander den Befehl »B laden«. Eine automatische Optimierung ist deshalb schwierig, weil sie auch absichtlich programmierte Seiteneffekte berücksichtigen müßte.

- Bei der Übergabe von Parametern wird als allererstes eine »Lokalkopie« erzeugt, weil der Compiler nicht weiß, ob der entsprechende Wert gleich bei der nächsten Operation gebraucht wird. Ähnliches gilt für das Zurückliefern von Funktionsergebnissen.

Alle drei Punkte laufen auf dasselbe hinaus: Durch eine manuelle Optimierung und (manchmal) viel Denkarbeit lassen sich Lade- und Speicheroperationen einsparen. Bei einer Befehlsfolge wie der Mandelbrot-Iteration, die praktisch nur aus »kurzen« Fließkomma-Operationen besteht, ist der 8087 mit FSTP- und FLD-Befehlen tatsächlich so beschäftigt, daß ihre Eliminierung einen Geschwindigkeitszuwachs von ziemlich genau 100% erbringt - ein Gewinn, der in einem normalen Programm ein gutes Stück geringer ausfallen dürfte.

Was den Gesamtenergieaufwand angeht - es soll schon mancher ganze Wochenenden mit einer Optimierung verbracht haben, die im Endeffekt mehrere Sekunden Laufzeit einspart. (Die Mandelbrot-Iteration, bei der wirklich jede Mikrosekunde zählt, stellt auch in diesem Punkt eine einsame Ausnahme dar).

---

*) Ganz im Gegensatz zu TURBO87 (der 8087-Variante der Version 3.0), Turbo Basic und einer Reihe anderer Compiler, die nur die beiden obersten Register des 8087 benutzen und deshalb um das vier- bis achtfache langsamer sind als ein gut durchdachtes handgeschriebenes Programm.

Ein **Hinweis:** In Kapitel 4 wird das Programm TRACE287 vorgestellt, mit dem sich (zusammen mit DEBUG oder SYMDEB) die Wirkung von Coprozessor-Befehlen schrittweise verfolgen läßt.

Apfelmännchen mit Köpfchen

Ich hatte einmal auf dem Apple][eine 6502-Routine entwickelt, die ein Array von Applesoft-Strings mit (vergleichsweise) wahnwitziger Geschwindigkeit absucht. Alle paar Wochen wurden einige weitere Millisekunden aus dieser Routine herausgekürzt - solange, bis ich zum erstenmal etwas über den Shell/Metzner-Algorithmus zu lesen bekam, der bei einer sortierten Liste mit 1000 Elementen nicht 1000 Vergleiche benötigt, sondern lächerliche 10.

Was eine Geschichte aus dem mittleren Tertiär der Computerei in einem Buch über Turbo Pascal 4.0/5.0 zu suchen hat? Nun - am Grundsatz, daß man einen Algorithmus mindestens dreimal überdenken sollte, bevor man anfängt, »Mikrosekunden zu putzen«, hat sich nicht viel geändert (genausowenig wie an der Zahl der Programmierer, die sich inzwischen daran halten).

Der Mathematiker Werner Durandi hat in der Ausgabe 3/87 der Zeitschrift c't erst einmal gezeigt, wann die Mandelbrot-Iteration *überhaupt nicht gebraucht wird* - und sich dann in einem weiteren Artikel über die Mikrosekunden hergemacht*). Die Endergebnisse seiner Überlegungen sind:

- Der »Kopf« des Apfelmännchens besteht aus einem Kreis mit dem Radius 1/4 um den Punkt mit dem Realteil -1 und dem Imaginärteil 0.
- Der »Körper« besteht aus einer sogenannten Epizykloide um den Punkt mit dem Realteil 0 und dem Imaginärteil 0. Die Kreislinie dieser Zykloide (d.h. die Begrenzung des »Körpers«) entsteht, wenn man einen Kreis mit dem Radius 1/4 auf einem anderen Kreis mit demselben Radius abrollt und dabei einen Punkt auf dem äußeren Kreis verfolgt. Am Anfang der Bewegung befindet sich der zu verfolgende Punkt auf der Linie des inneren Kreises (Abstand zum Punkt (0,0) = 1/4). Der maximale Abstand (3/4 = Radius des inneren Kreises plus Durchmesser des äußeren Kreises) ergibt sich, wenn die Kreisumdrehung zur Hälfte ausgeführt ist - an diesem Punkt stoßen »Kopf« und »Körper« des Apfelmännchens aneinander. Wenn die Bewegung fortgesetzt wird, verringert sich der Abstand wieder, bis wir nach einer vollständigen Umdrehung wieder beim Abstand 1/4 (und damit am Ausgangspunkt) angekommen sind.

Das Apfelmännchen ist durch diese beiden Kreise nicht vollständig abgegrenzt, an den Kreisrändern finden sich noch einige weitere Löcher (oder Auswüchse, wie man will) - es läßt sich aber mathematisch beweisen, daß die Iteration für Punkte *innerhalb* dieser beiden Begrenzungslinien mit Sicherheit erfolglos endet.

*) Wenn Sie diese beiden Artikel kennen, werden Sie auf den folgenden Seiten nicht viel Neues finden (abgesehen von einer Anpassung an die Versionen 4.0 bzw. 5.0).

Wenn wir ein Koordinatenpaar als Hypotenuse eines rechtwinkligen Dreiecks interpretieren, dann läßt sich mit

```
if Sqrt( Sqr(Realteil+1) + Sqr(Imaginärteil)) <= 1/4 ...
```

prüfen, ob der entsprechende Punkt innerhalb des »Kopfkreises« liegt. Diese Ungleichung läßt sich auch einfacher formulieren:

```
if Sqr(Realteil+1) + Sqr(Imaginärteil) <= 1/16 ....
```

Die Prüfung, ob der Punkt innerhalb des »Körpers« liegt, fällt etwas komplexer aus, weil hier sowohl der innere als auch der äußere Kreis berücksichtigt werden müssen. Der innere Kreis ergibt sich wie beim »Kopf« - nur daß hier der Mittelpunkt auf dem Punkt (0,0) liegt:

```
r := Sqr(Realteil) + Sqr(Imaginärteil);
```

Die Grundlagen der Berechnung des äußeren Kreises übersteigen nicht nur den Rahmen dieses Kapitels, sondern auch meine Kenntnisse der Trigonometrie. Ihr Pascal-Äquivalent nach der Berechnung von *r* ist:

```
s := Sqrt(r - Realteil/2 + 1/16);
if 16 * r * s <= 5 * s - 4 * Realteil + 1 ....
```

Wenn auch nur einer der beiden Tests *True* ergibt, dann liegt der Punkt innerhalb des Apfelmännchens und braucht nicht berechnet zu werden. Die Rechenzeit für den »Standardausschnitt« (d.h. die Gesamtansicht des Apfelmännchens) verringert sich durch diesen Kreis-/Zykloidentest auf rund ein Viertel.

Die Pascal-Version

Abgesehen vom Kreis-/Zykloidentest enthält das Programm APFEL1.PAS zwei Besonderheiten: eine Skalierung, die einigermaßen unabhängig vom verwendeten Grafik-Adapter arbeitet, und einen »Vergröberungsfaktor« (Konstante *MAG*), über den sich die Rechenzeit zum Ausprobieren von Bildausschnitten beliebig verkürzen läßt. Für MAG = 1 wird jeder Punkt berechnet und gezeichnet, für MAG = 2 nur noch jeder zweite Punkt auf jeder zweiten Zeile usw.

MAG macht einige zusätzliche Multiplikationen und den Einsatz von *while* anstelle (schnellerer) *for*-Schleifen notwendig, der zeitliche »Overhead« ist aber gegenüber der eigentlichen Berechnung vernachlässigbar. Für ein EGA-Bild mit 640*350 Punkten, einer Rechentiefe von 160 und MAG = 1 ergeben sich die folgenden Einzelzeiten in Sekunden:

```
Aufrufe von Iterate (+Parameterübergabe):     6
while-Schleifen, Test auf KeyPressed:        17
Zeichnen der Punkte mit PutPixel             20
Kreis-/Zykloidentest                        108
Iteration selbst                            557
                                           ----
                        Gesamt:             708  Sekunden
```

Der Zeitaufwand für das Zeichnen der Punkte ist erstaunlich hoch - hier lassen sich mit direkten Zugriffen auf die EGA-Register noch einmal recht einfach um die 10 Sekunden herausholen. Eine Programmversion, die ohne MAG und mit *for*-Schleifen arbeitet, spart rund 7 Sekunden ein; die Entfernung des Tests über *KeyPressed* bringt (außer eventuellem Ärger, weil sich das Programm dann nur noch über RESET vorzeitig abbrechen läßt) so gut wie nichts, weil diese Prüfung nur am Ende jeder »Bildschirmzeile« stattfindet - insgesamt also 350 Mal.

```
program Apfell;
{$N-,R-,S-}              { Zeitvergleich für EGA, 12 MHz AT, MAG = 5 }
                         { ohne 80x87: 89 Sek }
uses Crt, Graph;         { mit  80x87: 38 Sek }
                         { Overhead mit MAG = 5, 8087: rund 6 Sekunden }
type
  Extended = Real;       { <- Ausklammern, wenn ein 8087 vorhanden ist }
                         {    und außerdem $N+ setzen }
var
  GraphDriver, GraphMode: Integer;

  ReStep, ImStep: Extended;  { Schrittweite für Real- und Imaginärteil }
  RePart, ImPart: Extended;  { momentane Werte für Real-/Imaginärteil }
  Zr, Zs, ImQuad: Extended;  { für den Kreis-/Zykloidentest }
  x,y: Integer;              { momentane Koordinate }
  XAsp, YAsp: Word;        { Seiten-/Höhenverhältnis des Bildschirms }
  MaxX, MaxY, MaxC, CStep: Integer;
  Farbe: Integer;

const
  ReStart = -2.267; ReEnd = 1.0;     { Realteil von -2.657 bis 1.0 }
  ImStart = -1.125;                  { Imaginärteil }
  MAX: Integer = 160;        { maximale Iterationstiefe }
  MAG = 5;          { Vergröberungsfaktor - spart 2500% Zeit }

function Iterate(Re,Im: Extended): Integer;
var
  x,y,x2,y2: Extended;
  k: Integer;
begin
  x:= 0; y:= 0; x2 := y; y2 := 0; k:= 0;
  repeat
    y:= 2 * x * y + Im; x:= x2 - y2 + Re;
    x2 := Sqr(x); y2 := Sqr(y);
    Inc(k);
  until (x2 + y2 > 4) or (k >= MAX);
  Iterate := k;
end;
```

```
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode,'');
  GetAspectRatio(XAsp,YAsp);
  MaxX := GetMaxX; MaxY := GetMaxY;
  MaxC := MaxColors + 1; CStep := MAX div (MaxC);
  ReStep := (ReEnd-ReStart) / MaxX;  { Schrittweite Realteil }
  ImStep := ReStep * YAsp / XAsp;    { Schrittweite Imaginärteil }

  y := 0;
  ImPart := ImStart;
  while y <= MaxY do        { Der Imaginärteil bleibt jeweils für eine }
  begin                     { Bildschirmzeile gleich }
    RePart := ReStart;
    ImQuad := Sqr(ImPart);
    x := 0;
    while x <= MaxX do
    begin
      { Der Kreis-/Zykloidentest nach Werner Durandi (c't 3/87) }
      Zr := Sqr(RePart) + ImQuad;
      Zs := Sqrt(Zr - 0.5 * RePart + 0.0625);
      if (16.0 * Zr * Zs > 5 * Zs - 4 * RePart + 1)
            and (Sqr(Repart+1) + ImQuad > 0.0625)
       then Farbe := Iterate(RePart,ImPart)
       else Farbe := MAX;
      PutPixel(x,y,MaxC * CStep - Farbe);
      Inc(x, MAG); RePart := RePart + ReStep * MAG;  { Realteil erhöhen }
    end; { while x }

    if KeyPressed then
    begin                        { Nach jeder Zeile eine Prüfung, ob der }
      Readln;                    { Zuschauer die Lust verloren hat }
      CloseGraph;
      Exit;
    end;
    Inc(y, MAG);
    ImPart := ImPart + ImStep * MAG;
  end; { while y }

  Readln;
  CloseGraph;
end.
```

Der Ersatz von Iterate durch eine Maschinenroutine...
beschleunigt die Ausführung (wie angekündigt) noch einmal um das Doppelte*). In der 4.0-Version von APFEL1.PAS ändert sich nur die Deklaration dieser Funktion:

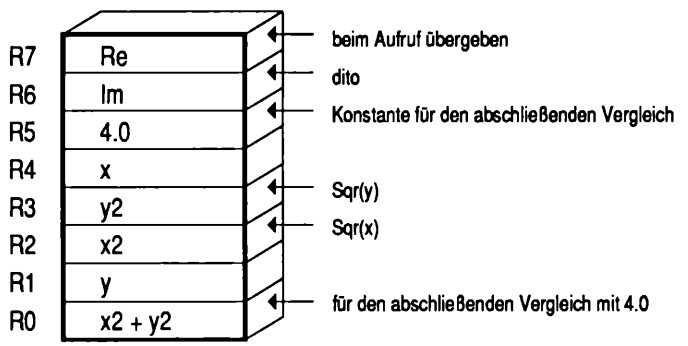
```
function Iterate(Re,Im: Extended): Integer; external;

{$L ITER8087}
                    { <- Rumpf der Funktion entfällt }
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode,'');
    ....
```

Achtung: das Programm muß auf jeden Fall Variablen des Typs *Extended* verwenden - mit *Real* kann die Maschinenversion von *Iterate* nichts anfangen. Wer die entsprechende **type**-Deklaration nicht aus APFEL1.PAS hinauswirft, erhält Wunderliches in Form zweier Meldungen:

```
Floating point error at xxxx:yyyy
Module has no debug information (APFEL2.PAS)
```

(Was uns der Dichter mit der zweiten Zeile sagen will: Der Fließkomma-Fehler ist innerhalb des Maschinenprogramms ITER8087.OBJ aufgetreten, das - im Gegensatz zu APFEL2.PAS - keine Informationen zur Fehlersuche enthält).

Turbo Pascal übergibt die Werte von *Re* und *Im* beim Aufruf von *Iterate* über den Stack des 8087, der innerhalb der Funktion folgendermaßen belegt ist:



Der Stack des 8086 wird nur zur Speicherung des Vergleichsergebnisses ($x2+y2>4$) benutzt, das Register BX zählt die Schleifendurchläufe und ersetzt so die Variable *k* der Pascal-Version. Obwohl *Iterate* eine Funktion ist, wird kein Platz für die Zwischenspeicherung des Ergebnisses reserviert. Sämtliche Zwischenergebnisse bleiben in den Registern des 8087.

*) Um genau zu sein: 266 anstelle von 557 Sekunden für ein EGA-Bild mit MAG = 1.

Ein Hinweis: Die notwendigen WAIT-Befehle werden von MASM automatisch vor jedem 8087-Befehl eingefügt und erscheinen deshalb nicht im Quelltext.

```
; ITER8087 - Mandelbrot-Iteration im direkten Verfahren
; Original von Werner Durandi (c't 3/87), für 4.0/5.0 angepaßt
; und leicht modifiziert von as, 08/01/88
        .8087
DATA    Segment Public Word
        EXTRN Max: Word             ; Obergrenze (Pascal-Variable)
DATA    ends

CODE    Segment Public Byte
        Public Iterate
        ASSUME cs:CODE, ds:DATA
; ---------------------------------------------------
; function Iterate(RePart, ImPart: Extended): Integer;
; ---------------------------------------------------

Four    dt 4.0                  ; Konstante .4E+1
Iterate Proc near
        push bp
        mov bp,sp
        fld Four                ; 4.0 -> R5
        xor bx,bx               ; k := 0;
        fldz                    ; x2 (R4) := 0;
        fldz                    ; y2 (R3) := 0;
        fldz                    ; x (R2) := 0;
        fldz                    ; y (R1) := 0;

; y := 2 * x * y + ImPart;
        fmul st,st(3)           ; y := y * x
Iter:   fadd st,st(0)           ; * 2
        fadd st,st(5)           ; + ImPart

; x := x2 - y2 + RePart;
        fincstp                 ; auf R2 (x)
        fsub st,st(1)           ; x2 - y2
        fadd st,st(5)           ; + RePart
        fst st(2)               ; ==> x (R4)

; x2 := Sqr(x);
        fmul st,st(0)           ; ==> x2 (R2) := Sqr(x);
        push ax                 ; Platz für FSTSW
        inc bx                  ; Inc(k);
        fst st(6)               ; x2 in R0 (für x2 + y2 > 4.0)
```

```
; y2 := Sqr(y);
        fdecstp                 ; ST auf R1
        fst st(2)               ; y2 := y
        fmul st(2),st           ; y * y ==> y2 (R3)

; until (x2+y2 > 4.0) or (k >= max);
        fdecstp                 ; ST auf R0
        fadd st,st(3)           ; x2 + y2
        fcomp st(5)             ; ==> if x2 + y2 > 4.0 (R5), ST danach auf R1
        fstsw Word Ptr [bp-2]   ; Ergebnis des Vergleichs
        fmul st,st(3)           ; y (R3) := y (R1) * x (R4) für Wiederholung
        pop ax                  ; Statuswort (von fstcw) -> AX
        sahf                    ; AH -> Flags
        ja Done                 ; ==> größer 4.0 -> Ende
        cmp bx,[MAX]            ; Zähler = MAX?
        jb Iter
Done:   mov ax,bx               ; k als Funktionsergebnis
        mov sp,bp               ; eigentlich unnötig
        pop bp
        finit                   ; alle 8087-Register entleeren!
        ret
Iterate endp
CODE    ends
        end
```

Vorsicht! Sämtliche Register des 8087 sind am Ende der Routine noch besetzt. Der Befehl FINIT ist zwar die einfachste Möglichkeit, sie wieder zu entleeren, setzt aber auch das Kontrollwort-Register neu - und verhindert damit eine Reaktion der Laufzeitbibliothek auf Divisionen durch Null, ungültige Befehle usw. Nötigenfalls müssen Sie die Masken-Bits dieses Registers direkt neu setzen (siehe »Geschwindigkeit und Präzision«).

Version 5.0

Da die Maschinenversion von *Iterate* auf »Lokalkopien« verzichtet und die übergebenen Parameter direkt in den Registern des 8087 erwartet, gäbe es hier zwei Möglichkeiten: Entweder

❑ erweitert man ITER8087.ASM um entsprechende Ladebefehle, also

```
; ---------------------------------------------------
; function Iterate(RePart, ImPart: Extended): Integer;
; ---------------------------------------------------

Four    dt 4.0                  ; Konstante .4E+1
Iterate Proc near
        push bp
        mov bp,sp
```

```
IF TP50
        fld TByte Ptr[bp+04]  ; RePart -> R7
        fld TByte Ptr[bp+14]  ; ImPart -> R6
ENDIF
        fld Four              ; 4.0 -> R5
        xor bx,bx             ; k := 0;
        fldz                  ; x2 (R4) := 0;
        fldz                  ; y2 (R3) := 0;
        ....
```

und ruft MASM in der Form `MASM /DTP50 ITER8087;` auf, nimmt in Kauf, daß es zwei verschiedene .OBJ-Versionen dieser Routine gibt; versucht weiterhin, die rund 4 zusätzlichen Millisekunden pro Aufruf zu verschmerzen

❑ oder man baut sich ein (nicht so hundertprozentig astreines) Hintertürchen und überläßt die Sache dem Compiler. Diese Lösung besteht aus zwei Anweisungen zur bedingten Compilierung im Quelltext:

```
{$IFDEF VER40}
function Iterate(Re,Im: Extended): Integer; external;
{$ELSE}
function Iterate: Integer; external;
{$ENDIF}
{$L Iter8087}
....
....
      { Der Kreis-/Zykloidentest nach Werner Durandi (c't 3/87) }
      Zr := Sqr(RePart) + ImQuad;
      Zs := Sqrt(Zr - 0.5 * RePart + 0.0625);
      if (16.0 * Zr * Zs > 5 * Zs - 4 * RePart + 1)
           and (Sqr(Repart+1) + ImQuad > 0.0625)
{$IFDEF VER40}
       then Farbe := Iterate(RePart,ImPart)
{$ELSE}
       then begin  { für die Version 5.0: direktes Laden der 8087-Register }
         inline ($CD/$37/$2E/RePart/     { INT $37 ; FLD TByte Ptr [RePart] }
                 $CD/$37/$2E/ImPart);    { INT $37 ; FLD TByte PTr [ImPart] }
         Farbe := Iterate;   { Aufruf ohne Parameter }
       end
{$ENDIF}
      else Farbe := Max;
      PutPixel(x,y,MaxC * CStep - Farbe);
```

Wie gesagt - der Weisheit letzter Schluß ist eine Funktion, die ihre Deklaration je nach Compilerversion ändert, sicher nicht. Dafür sparen wir uns aber nicht nur eventuelle Zusatzparameter beim Aufruf von MASM, sondern auch einige Milli-

sekunden, die ansonsten bei jedem Aufruf von *Iterate* durch letztlich unnötiges Hin- und Herkopieren der Parameter »verbraten« würden*).

Achtung: Bei der Übersetzung von ITER8087.ASM mit dem Befehl

```
MASM /E ITER8087;
```

erzeugt MASM »emulator-kompatible« Coprozessorbefehle, d.h. verwandelt jeden 8087-Opcode in einen $3x-Interrupt. Theoretisch wäre APFEL2.PAS mit einer nachfolgenden Compilierung im Modus {$N+,E+} damit auch auf Computern ohne Coprozessor lauffähig - praktisch endet der Versuch bedauerlicherweise an dem einzigen Punkt, in dem der Emulator den Coprozessor *nicht* korrekt simuliert: Wie auch im Handbuch dokumentiert, werden kreisförmige Adressierungen der Rechenregister nicht unterstützt - und genau auf dieser Adressierungsform basiert der größte Teil der Optimierungen von ITER8087. Die Codierung der Parameterübergabe als Emulator-Interrupt ist deshalb rein kosmetischer Natur - zumindest bis zur nächsten Version des Compilers.

Der Ersatz des Kreis-/Zykloidentests durch eine Maschinenroutine... reduziert die Rechenzeit für ein EGA-Bild mit MAG = 1 von 416 auf 385 Sekunden, spart also noch einmal eine halbe Minute. Der Hauptgrund für die mäßige Ausbeute liegt darin, daß eine so komplexe Operation wie der Vergleich

```
if (16 * r * s > 5 * d - 4 * Re + 1) and (Sqr(Re+1) + Sqr(Im) > 1/16) ...
```

vom Compiler bereits fast optimal umgesetzt wird - mit manueller Optimierung läßt sich gerade noch eine Steigerung um den Faktor 1.4 (77 anstelle von 108 Sekunden) herausholen. Die beiden folgenden Listings stellen deshalb hauptsächlich eine Rechtfertigung zuvor hineingesteckter Arbeit dar.

Version 5.0

Die für die Version 5.0 notwendigen Befehle zur bedingten Compilierung sind im Listing von APFEL3.PAS mit enthalten (wobei hier mit demselben Hintertürchen wie bei APFEL2 gearbeitet wird). Für die Maschinenroutine CYCLTEST gilt wegen dieses Hintertürchens dasselbe wie für ITER8087 - sie läßt sich für beide Versionen unverändert verwenden.

```
program Apfel3;          { Zeitvergleich (12 MHz 80286, 8 MHz 80287): }
{$N+,R-,S-}              { CGA, MAG = 1   - 1 Min, 48 Sek }
                         { EGA, MAG = 5:                  }
                         { ohne 80x87:          89 Sek    }
uses Crt, Graph;         { mit  80x87:          38 Sek    }
                         { mit ITER87.OBJ       16 Sek    }
                         { mit CYCLTEST.OBJ     15 Sek    }
```

*) Und weil es hier zum einen um Geschwindigkeit und zum anderen um »Nachbausicherheit« geht, findet sich diese Version des Programms auch auf der Diskette.

```
var
  GraphDriver, GraphMode: Integer;

  ReStep, ImStep: Extended;  { Schrittweite für Real- und Imaginärteil }
  RePart, ImPart: Extended;  { momentane Werte für Real/Imaginärteil }
  Zr, Zs, ImQuad: Extended;  { für den Kreis-/Zykloidentest }
  x,y: Integer;              { momentane Koordinate }
  XAsp, YAsp: Word;        { Seiten-/Höhenverhältnis des Bildschirms }
  MaxX, MaxY, MaxC, CStep: Integer;
  Farbe: Integer;

const
  ReStart = -2.267; ReEnd = 1.0;     { Realteil von -2.657 bis 1.0 }
  ImStart = -1.125;                  { Imaginärteil }
  MAX: Integer = 160;        { maximale Iterationstiefe }
  MAG = 5;          { Vergröberungsfaktor - spart 2500% Zeit }
  Z_SIX: Extended = 0.0625;

{$IFDEF VER40}
function Iterate(Re,Im: Extended): Integer; external;
function CyclTest(ZSIX, Re, ImQ: Extended): Boolean;  external;
{$ELSE}
function Iterate: Integer; external;
function CyclTest: Boolean; external;
{$ENDIF}

{$L ITER8087}
{$L CYCLTEST}
```

Die Definition des Wertes 0.0625 als typisierte Konstante ist für die Version 4.0 nicht notwendig - dort könnte man diesen Wert beim Aufruf von *CyclTest* auch direkt (d.h. untypisiert) angeben. Die im Listing gezeigte »5.0-kompatible« Variante von *CyclTest* hat dieselbe Krankheit wie *Iterate* - beide Funktionen ändern ihre Deklaration je nach Version des Compilers.

```
begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode,'');
  GetAspectRatio(XAsp,YAsp);
  MaxX := GetMaxX; MaxY := GetMaxY;
  MaxC := MaxColors + 1; CStep := MAX div (MaxC);
  ReStep := (ReEnd-ReStart) / MaxX;  { Schrittweite Realteil }
  ImStep := ReStep * YAsp / XAsp;    { Schrittweite Imaginärteil }
  y := 0;
  ImPart := ImStart;
```

```
  while y <= MaxY do        { Der Imaginärteil bleibt jeweils für eine }
  begin                     { Bildschirmzeile gleich }
    RePart := ReStart;
    ImQuad := Sqr(ImPart);
    x := 0;
    while x <= MaxX do
{$IFDEF VER40}
      if not CyclTest(0.0625, RePart, ImQuad)
       then Farbe := Iterate(RePart,ImPart)
       else Farbe := MAX;
```

So klar, einfach und übersichtlich hat die gesamte Berechnung in der Version 4.0 einmal ausgesehen. Damit die Sache auch noch für die Version 5.0 funktioniert, sind dagegen einige Verrenkungen notwendig:

```
{$ELSE}              { für die Version 5.0: direktes Laden der 8087-Register }
        inline ($CD/$37/$2E/Z_SIX/      { INT $37 ; FLD TByte Ptr [Z_SIX]  }
               $CD/$37/$2E/RePart/      { INT $37 ; FLD TByte Ptr [RePart] }
               $CD/$37/$2E/ImQuad);     { INT $37 ; FLD TByte PTr [ImQuad] }

        if not CyclTest   { Aufruf ohne Parameter }
         then begin  { für V 5.0: direktes Laden der 8087-Register }
          inline ($CD/$37/$2E/RePart/   { INT $37 ; FLD TByte Ptr [RePart] }
                 $CD/$37/$2E/ImPart);   { INT $37 ; FLD TByte PTr [ImPart] }
          Farbe := Iterate;   { Aufruf ohne Parameter }
         end
        else Farbe := MAX;
{$ENDIF}

      PutPixel(x,y,MaxC * CStep - Farbe);

      Inc(x, MAG); RePart := RePart + ReStep * MAG;  { Realteil erhöhen }
    end; { while x }
    if KeyPressed then
    begin                        { Nach jeder Zeile eine Prüfung, ob der }
      Readln; CloseGraph; Exit;  { Zuschauer die Lust verloren hat }
    end;
    Inc(y, MAG); ImPart := ImPart + ImStep * MAG;
  end; { while y }
  Readln;
  CloseGraph;
end.
```

Die Funktion *CyclTest* liefert *True* zurück, wenn sich der Punkt innerhalb des Kreises befindet. Sie bekommt beim Aufruf drei Parameter übergeben: die Konstante 0.0625 (= 1/16), den Realteil und das Quadrat des Imaginärteils.

Was nun (auf der nächsten Seite) noch folgt, ist das Listing der Maschinenversion von *CyclTest*:

```
; CYCLTEST - Kreis-/Zykloidentest im direkten Verfahren
; Original von Werner Durandi (c't 3/87), für 4.0/5.0 angepaßt
; und recht kräftig modifiziert von as, 08/01/88

        .8087
CODE    Segment Byte public
        PUBLIC CyclTest
        assume cs:CODE
MOFIVE  dt -0.5             ; die diversen Konstanten, die für
FOUR    dt 4.0              ; den Kreis-/Zykloidentest gebraucht werden
FIVE    dt 5.0
SIXTEEN dt 16.0

; ---------------------------------------------------
; function CyclTest(ZSIX, Re, ImQ: Extended): Boolean;
; ---------------------------------------------------

CyclTest proc near
        push bp
        mov bp,sp
        sub sp,3
        fld1                ;                       0.0625
        fadd st,st(2)       ; RePart + 1.0          RePart
        fmul st,st(0)       ; Sqr(RePart + 1.0)     ImQ
        fadd st,st(1)       ; + Sqr(ImPart)         1.0
        fcomp st(3)
        fstsw [bp-3]
        mov ax,[bp-3]
        sahf
        jbe true            ; -> innerhalb des "Kopfkreises"

        fld st(1)           ; RePart
        fmul st,st(0)       ; Sqr(RePart)           0.0625
        faddp st(1), st     ; + ImQ -> Zr           ImQ
        fld MOFIVE          ;                       RePart
        fmul st,st(2)       ; RePart * -0.5         1.0
        fadd st,st(3)       ; + 0.0625
        fadd st,st(1)       ; + Zr  -> Zs           Zs
        fsqrt
        fld SIXTEEN
        fmul st,st(1)       ; 16 * Zs
        fmulp st(2),st      ; 16 * Zs * Zr  -> Zr
        fld FIVE
        fmulp st(1),st      ; 5 * Zs
        fld1
        faddp st(1),st      ; + 1.0
        fld FOUR
        fmulp st(3),st      ; RePart * 4.0
```

```
        fsub st,st(2);    ; 5 * Zs + 1 - 4 * RePart
        fcom st(1)        ; Vergleich mit 16 * Zr * Zs
        fstsw[bp-4]
        mov ax,[bp-4]
        sahf
        mov ax,0000
        jb false
true:   mov ax,1          ; innerhalb der Zykloide
false:  finit
        mov sp,bp
        pop bp
        ret
CyclTest endp
CODE    ends
        end
```

Kapitel 3
Zeiger und Rekursion

Nach rund 160 Seiten Detail-Bombardement geht es in diesem Kapitel fast ausschließlich um Pascal und damit um die reine Wissenschaft an sich. Ob die Reihenfolge der Kapitel etwas durcheinander gekommen ist? Nein - es hat sich beim Schreiben dieses Buches herausgestellt, daß es andersherum nicht funktionieren *kann*: Wer sich bei der Beschreibung komplizierter logischer Konstrukte wie Baumstrukturen ständig um die Frage herumdrücken muß, wieso lokale Variablen auf geisterhafte Weise immer wieder neu zur Verfügung stehen, wird (meiner Meinung nach) mit jedem Erklärungsversuch scheitern.

Dasselbe gilt für Zeigervariablen: Solange keine konkrete Vorstellung vorhanden ist, *wohin* ein Zeiger überhaupt zeigen kann, schlägt man sich im leeren Raum mit syntaktischen Details herum.

So. Nach diesen Entschuldigungen können wir wohl anfangen.

Zeigervariablen

»Ein Zeiger enthält eine *Adresse* und zeigt deshalb auf einen Wert, anstatt ihn direkt zu enthalten.« So oder ähnlich lesen sich Definitionen dieses Variablentyps in den meisten Pascal-Handbüchern. Dazu kommt dann noch die absolut logische, aber recht komplizierte Syntax von Pascal - und selbst ausgebuffte Maschinenprogrammierer flüchten sich angesichts von Ausdrücken wie

```
PtrArray[x]^        { die Variable, auf die der x.Zeiger von PtrArray zeigt }
 := ArrayPtr^[y]; { das Element y des Arrays, auf das ArrayPtr zeigt }
```

in die abenteuerlichsten Umwege. Dabei ist die Sache wirklich furchtbar einfach, wenn man die Formalitäten von Pascal einmal außer acht läßt und sich ansieht, welche Befehle der Compiler für Zeigeroperationen erzeugt*). Ein Beispiel:

```
var x: Integer;
    IntPtr: ^Integer;  { Zeiger auf einen beliebigen Integerwert }
begin
  IntPtr := @x;    { Zeiger auf die Adresse von x setzen }
  IntPtr^ := 4;    { ??? }
  Writeln(x);      { ergibt 4 }
end.
```

*) Beim Autor ist der berühmte Groschen nach rund 3 Jahren Programmierpraxis mit Pascal gefallen - während der Analyse der Laufzeitbibliothek von Turbo 3.0.

Der Compiler reserviert hier die Speicherzellen 0000/01 des Datensegments für den Wert der Variablen *x* (d.h. übersetzt eine Leseoperation von *x* wie bei *Writeln* mit »Lese den Inhalt der Speicherzellen 0000/01«). Die Variable *IntPtr* ist auf dieselbe Weise mit den Speicherzellen 0002..05 des Datensegments assoziiert.

Ein **Hinweis zur Version 5.0:** Die erste globale Variable des Programms (*x*) hat hier die Adresse $003C, die Adressen der Prozessorbefehle sind wegen des PUSH BP zu Beginn des Hauptprogramms jeweils um zwei Bytes verschoben (vgl. Kapitel 1). Abgesehen von diesen beiden Punkten sind die erzeugten Programme bis aufs letzte Bit gleich - und wir können uns eine Verdoppelung des Kapitelumfangs sparen.

Der Compiler übersetzt die Zuweisung `IntPtr := @x;` in vier Prozessorbefehle:

```
0007 B80000        MOV    AX,0000              ; Adresse von X (Offset)
000A 8CDA          MOV    DX,DS                ; Adresse von X (Segment)
000C A30200        MOV    [INTPTR],AX
000F 89160400      MOV    [INTPTR+02],DX       ; IntPtr := DS:0000
```

Danach ist die *Anfangs*adresse von *x* (Datensegment:0000) in der Variablen *IntPtr* gespeichert. Ein »Zugriff über *IntPtr*« besteht nun aus zwei Schritten:

- die Variable *IntPtr* wird gelesen - damit erhalten wir in unserem Beispiel die Adresse, ab der die Variable *x* gespeichert ist;
- die Speicherzellen, die sich auf der soeben ermittelten Adresse befinden, werden neu beschrieben oder gelesen - je nach Art des Zugriffs.

Tatsächlich lassen sich diese beiden Schritte mit zwei Prozessorbefehlen erledigen:

```
** IntPtr^ := 4;    { ??? }
0013 C43E0200       LES    DI,[INTPTR]           ; DS:0000 -> ES:DI
0017 26C7050400     MOV    Word Ptr ES:[DI],0004 ; 0004 -> DS:0000/01
```

Der Prozessor lädt hier also zuerst ein Registerpaar (ES:DI) mit dem Inhalt von *IntPtr* (= Datensegment:0000, die Adresse der Variablen *x*) und schreibt dann den Wert 0004 in die Speicherzellen, deren Adresse in ES:DI enthalten ist. Der Zusatz `Word Ptr` weist den Prozessor an, zwei aufeinanderfolgende Speicherzellen mit dem neuen Wert zu setzen, und bringt uns zur nächsten Überschrift:

Variablengrößen und -strukturen

Laut Referenzhandbuch kann eine Zeigervariable entweder *typisiert* (mit `^<Variablentyp>`) oder »generisch« (als *Pointer*) deklariert werden. Im ersten Fall ist die Variable nur zu Zeigern kompatibel, die für denselben Variablentyp deklariert wurden; im zweiten Fall ist sie zu sämtlichen Zeigern kompatibel, läßt sich aber nicht direkt für Zugriffe verwenden. Wunderbar, sagt sich der geneigte Leser, der nicht sowieso schon weiß, was gemeint ist, sucht im *Tutorial* vergeblich nach

einem brauchbaren Anwendungsbeispiel, findet keins und wendet sich anderen Dingen zu*).

Auch hier ist die Erklärung am einfachsten, wenn man sich die vom Compiler erzeugten Prozessorbefehle ansieht. In einer Modifikation des zuerst gezeigten Programms, die *x* als *Byte* und *IntPtr* als *^Byte* deklariert, also

```
var
  x: Byte;
  IntPtr: ^Byte;  { Zeiger auf ein beliebiges Byte }

begin
  IntPtr := @x;   { Zeiger auf die Adresse von x setzen }
  IntPtr^ := 4;   { ??? }
  Writeln(x);     { ergibt 4 }
end.
```

wird die Zuweisung `IntPtr^ := 4;` mit den folgenden Prozessorbefehlen ausgeführt:

```
0013 C43E0100       LES    DI,[INTPTR]            ; DS:0000 -> ES:DI
0017 26C60504       MOV    Byte Ptr ES:[DI],04    ; 04 -> DS:0000
```

Der Zusatz `Byte Ptr` weist den Prozessor an, nur die Speicherzelle DS:0000 mit einem neuen Wert zu setzen.

Wenn wir *x* als *LongInt* und *IntPtr* als *^LongInt* deklarieren, also

```
var
  x: LongInt;
  IntPtr: ^LongInt;  { Zeiger auf einen beliebigen LongInt }

begin
  IntPtr := @x;   { Zeiger auf die Adresse von x setzen }
  IntPtr^ := 4;   { ??? }
  Writeln(x);     { ergibt 4 }
end.
```

dann ergeben sich für die Zuweisung `IntPtr^ := 4;` die Prozessorbefehle

```
0013 C43E0400       LES    DI,[INTPTR]            ; DS:0000 -> ES:DI
0017 26C7050400     MOV    Word Ptr ES:[DI],0004  ; 0004 -> DS:0000/01
001C 26C745020000   MOV    Word Ptr ES:[DI+02],0000 ; 0000 -> DS:0002/03
```

Hier wird gleich zweimal mit dem Zusatz `Word Ptr` gearbeitet, der Prozessor setzt also die Speicherzellen DS:0000..03.

Diese Erklärung hätte auch wesentlich kürzer (aber - denke ich jedenfalls - nicht so anschaulich) ausfallen können: *Der Compiler erkennt am Typ eines Zeigers, wieviele Speicherzellen bei Zugriffen gelesen bzw. gesetzt werden müssen.*

---

*) Im amerikanischen Original des Benutzerhandbuchs findet sich *überhaupt nichts* über die Anwendung von Zeigern - der entsprechende Teil des Kapitels 6 ist eine freie Erfindung von mir.

Das folgende Programm zeigt, daß dieser Grundsatz nicht nur für einfache Variablen gilt. Es deklariert zwei unterschiedliche Recordtypen und entsprechende Zeigervariablen:

```
program RecPointers;

type
  NumRec = record
             Line : string[31];  { belegt $20 Bytes }
             x: Integer;         { ** }
           end;
  RecNum = record
             x: LongInt;         { ** }
             Line : string[31];
           end;
var
  Rec1: NumRec;
  Rec1Ptr: ^NumRec;   { Zeiger auf einen Record des ersten Typs }
  Rec2: RecNum;
  Rec2Ptr: ^RecNum;   { Zeiger auf einen Record des zweiten Typs }

begin
  Rec1Ptr := @Rec1;   { Rec1Ptr auf die Adresse von Rec1 (DS:$0000) }
  Rec2Ptr := @Rec2;   { Rec2Ptr auf die Adresse von Rec2 (DS:$0022) }

  Rec1Ptr^.x := 4;   { ** Zuweisung 1 }
  Rec2Ptr^.x := 4;   { ** Zuweisung 2 }
  Writeln(SizeOf(Rec1Ptr^));  { ergibt 34 }

  Writeln(Rec1.x);   { ergibt 4 }
  Writeln(Rec2.x);   { dito }
end.
```

Die Zuweisungen über die Zeiger *Rec1Ptr* und *Rec2Ptr* ergeben die folgenden Prozessorbefehle:

```
** Rec1Ptr^.x := 4;
001F C43E2200        LES    DI,[REC1PTR]                ; @Rec1 -> ES:DI
0023 26C745200400    MOV    Word Ptr ES:[DI+20],0004    ; 0004 -> Feld x
** Rec2Ptr^.x := 4;
0029 C43E4A00        LES    DI,[REC2PTR]                ; @Rec2 -> ES:DI
002D 26C7050400      MOV    Word Ptr ES:[DI],0004       ; 0000 0004 -> Feld x
0032 26C745020000    MOV    Word Ptr ES:[DI+02],0000
```

Der Compiler hat hier nicht nur gemerkt, daß die Felder *x* innerhalb der beiden Recordvariablen an verschiedenen Stellen gespeichert sind, sondern weiß auch, daß *x* im ersten Fall für einen *Integer*, im zweiten Fall dagegen für einen *LongInt* steht.

Eine typisierte Zeigerdeklaration liefert dem Compiler dieselben Eckdaten wie eine Variablendeklaration entsprechenden Typs, also:

- die Größe der Variablen (auf die der Zeiger zeigt);
- die Struktur dieser Variablen (d.h. die relativen Adressen einzelner Felder, soweit vorhanden);
- die Größe der einzelnen Felder (soweit vorhanden).

Mit diesen Informationen ist er auch in der Lage, Typverletzungen zu erkennen - Anweisungen wie `RecIPtr^ := Rec2;` oder `RecIPtr := @Rec2;` werden als fehlerhaft zurückgewiesen.

Achtung! Dieser Schutzmechanismus funktioniert nur, solange die beiden Variablen nicht »zuweisungskompatibel« sind. Eine Adress-Ermittlung der Form *Int_Ptr := @Byte_Var* wird vom Compiler widerspruchslos hingenommen, weil Zuordnungen des Typs *Int_Var := Byte_Var* in Pascal zulässig sind. Das folgende Programm demonstriert die unerfreulichen Ergebnisse auf harmlose Weise:

```
program PointerCrash;
var
  x: Byte;
  IntPtr : ^Integer;

begin
  IntPtr := @x;       { IntPtr (Typ Integer) auf die Adresse von x (Typ Byte) }
  IntPtr^ := 10000;   { Zuweisung über IntPtr^ schreibt 2(!) Bytes "in x" }
  Writeln(IntPtr^);   { Ergebnis ist undefiniert }
end.
```

Für *x* ist hier ein einzelnes Byte im Datensegment reserviert (DS:0000), direkt darauf folgt der Speicherplatz für *IntPtr* (DS:0001..04). Der Compiler kann bei der Zuweisung `IntPtr^ := 10000;` nicht wissen, daß wir *IntPtr* zuvor auf eine *Byte*-Variable gesetzt haben, und läßt deshalb 2 Bytes schreiben:

```
** IntPtr^ := 10000;
0013 C43E0100       LES    DI,[INTPTR]            ; @x (= DS:0000) -> ES:DI
0017 26C7051027     MOV    Word Ptr ES:[DI],2710  ; setzt DS:0000/01 !!
```

x erhält hier den Wert $10 und das erste Byte des für *IntPtr* reservierten Speicherplatzes den Wert $27. *IntPtr* zeigt nach dieser versehentlichen Veränderung »irgendwohin« - und der Inhalt dieses »irgendwohin« wird dann von *Writeln* ausgegeben*).

*) Wie sich über eine .MAP-Datei feststellen läßt, gibt *Writeln* in diesem Programm die ersten beiden Bytes des *User*-Feldes von *Input* aus - diese Variable von *System* folgt (bei beiden Versionen des Compilers) im Datensegment unmittelbar auf die Variablen des Programms (vgl. Kapitel 1).

Generische Zeigertypen

Für die Deklaration einer Zeigervariablen als *Pointer* legt der Compiler recht spezielle »Eckdaten« fest:

- Größe der Variablen, auf die der Zeiger zeigt: nicht definiert;
- Struktur der Variablen: keine;
- Größe einzelner Felder: nicht definiert.

Eine Zuweisung wie `Pointer_Var := @x;` ist legal und speichert die Adresse von *x* in *Pointer_Var* - vor Zugriffen, die *Pointer*-Variablen *benutzen*, muß der Compiler aber erst über eine explizite Typ-Umwandlung Informationen erhalten, wieviele Bytes welcher Struktur bearbeitet werden sollen.

Der als »zu allen anderen typisierten Zeigern kompatibel« definierte Typ *Pointer* wurde hauptsächlich deshalb erfunden, weil Funktionen wie *GetMem* und *Ptr* (mit denen sich beliebige Zeiger setzen lassen) *irgendeinen* Ergebnistyp haben müssen. (Compiler, die den Typ *Pointer* nicht unterstützen, behelfen sich meist mit *^Byte* oder *^Char*).

Zum Thema »Notation«

Ohne eine vorherige Vereinbarung mit `type` akzeptiert der Compiler nur die Pascal-Grundtypen als »Ziel« für die Deklaration einer Zeigervariablen, also:

```
var
  Char_Ptr : ^Char;
  Byte_Ptr : ^Byte;
   ...
  LongInt_Ptr : ^LongInt;
  String_Ptr : ^string;
```

Der Versuch, eine Zeigervariable zusammen mit einem neuen strukturierten Typ zu deklarieren, beispielsweise

```
var
  Ary5_Ptr : ^array[1..5] of Integer;
```

wird in jedem Fall als unzulässig moniert. Wieso? Weil der Compiler auch getrennt deklarierte strukturierte Variablen wie in

```
var
  A: array[1..5] of Integer;
  B: array[1..5] of Integer;
```

als »inkompatibel« behandelt (vgl. Referenzhandbuch, Kapitel 14, Abschnitt »Identische und kompatible Typen«) - er geht deshalb bei einer Deklaration wie der von *Ary5_Ptr* davon aus, daß es innerhalb des restlichen Programms ja wohl nichts Verträgliches gibt, auf das dieser Zeiger zeigen könnte.

Diese absichtliche Einschränkung ist eigentlich nur dann lästig, wenn man auf Dinge außerhalb des Programms zugreifen will - wie beispielsweise den Bildspeicher:

```
type
  ScreenMem = array[0..1999] of record
                                  Ch    : Char;  { ASCII-Code }
                                  Attrib: Byte;  { Attribut }
                                end;
var
  Screen : ^ScreenMem; { Zeiger auf den Bildspeicher }
  x: Integer;
begin
  Screen := Ptr($B800,0);        { für Hercules: $B000,0 }
  for x:= 0 to 1999 do       { füllt den gesamten Bildschirm mit }
    Screen^[x].Ch := 'A';    { 'A's, Attribute bleiben unverändert }
end.
```

Mit `Screen^[x].Ch` sind wir beim eigentlichen Thema dieses Abschnitts angekommen*) - dieses Ungetüm bezeichnet das Feld *Ch* des *x*. Records des Arrays, dessen Startadresse in der Variablen *Screen* gespeichert ist.

Der Unterschied zwischen *Screen^[x].Ch* und einem (hypothetischen) Ausdruck *Screen[x]^.Ch* läßt sich am besten in einem »Referenzhandbuch-inkompatiblen« Satz ausdrücken: Wenn irgendwo in einem Variablenbezug ein ^ vorkommt, behandelt der Compiler alles links davon als Zeigervariable und alles rechts davon als Teil der Struktur, auf die diese Zeigervariable zeigt. Also:

`Screen^[x].Ch` der Compiler erwartet, daß *Screen* eine Zeigervariable auf ein Record-Array ist. Das Programm interpretiert den Inhalt von *Screen* als Startadresse des Arrays, addiert zuerst »*x* * Recordgröße« dazu und dann die relative Adresse des Feldes *Ch* innerhalb eines Records. Das Endergebnis ist die Adresse von *Ch* (bzw. der Wert dieses Feldes).

`Screen[x]^.Ch` der Compiler erwartet, daß es ein Zeiger-Array namens *Screen* gibt und daß die Zeiger dieses Arrays auf Records zeigen. Das Programm liest das *x*. Element dieses Arrays, interpretiert seinen Wert als Anfangsadresse eines Records und addiert die relative Adresse von *Ch* innerhalb des entsprechenden Recordtyps dazu. Das Endergebnis ist die Adresse von *Ch* (bzw. der Wert dieses Feldes).

`Screen[x].Ch^` der Compiler erwartet, daß es ein Record-Array namens *Screen* gibt und daß das Feld *Ch* dieser Records einen Zeiger enthält. Das Programm liest den *x*. Record dieses Arrays und aus die-

*) Wer den Unterschied zwischen `Screen^[x].Ch`, `Screen[x]^.Ch` und `Screen[x].Ch^` ohne jede Grübelei bestimmen kann, möge sich gratulieren und weiterblättern...

sem Record den Wert von *Ch*, der als Zeiger interpretiert wird. Das Endergebnis ist die Adresse, *die in* Ch *enthalten ist* (bzw. der Inhalt der Speicherzellen auf der Adresse, auf die *Ch* zeigt).

Eine letzte Komplikation, die vor allem bei verketteten Listen zum täglich Brot des Programmierers gehört (und auf die wir später noch zurückkommen): Zeiger, die auf Records mit Zeigern darin zeigen. Eine einfach verkettete Liste wie

```
Record1
  Daten
  Zeiger  -> Record2
               Daten
               Zeiger  -> Record 3
                            ...
```

macht eine Typ-Deklaration notwendig, bei der sich die Katze sozusagen selbst in den Schwanz beißt:

- um einen typisierten Zeiger auf einen Recordtyp *RecX* zu definieren, brauchen wir die Definition von *RecX*;
- *RecX* soll einen Zeiger auf den nächsten Record der Liste (der ebenfalls den Typ *RecX* hat) als Feld definieren - dazu brauchen wir aber einen Zeigertyp *^RecX*.

Damit derartige Deklarationen überhaupt möglich sind, macht Pascal hier eine Ausnahme von der sonst allgemeingültigen Regel, daß alle Bezeichner (und dazu gehören auch Variablentypen) *vor* ihrer Verwendung deklariert sein müssen: Die Definition eines typisierten Zeigers ist auch dann erlaubt, wenn der Compiler den »Zielvariablentyp« noch nicht kennt - vorausgesetzt, dieser Typ wird innerhalb *desselben* Typen-Deklarationsteils definiert. Ein Beispiel:

```
type
  RecXPtr = ^RecX;  { zu diesem Zeitpunkt kennt der Compiler den
                      Datentyp RecX noch nicht }
  RecX = record
           Daten: string[80];
           Next : RecXPtr;   { Zeiger auf weitere Records des Typs RecX }
         end;
```

Turbo Pascal läßt die Verwendung eines Datentyps erst dann zu, wenn seine Definition abgeschlossen ist. Eine Deklaration wie die folgende endet deshalb mit einer Fehlermeldung des Compilers:

```
type
  RecX = record
           Daten : string[80];
           Next  : ^RecX;    { "Undefined type in pointer definition(RECX)" }
         end;
```

(Prof. Wirth verwendet derartige Deklarationen recht häufig[1] - trotzdem wird dieser Unterschied auch im Vergleich mit dem ANSI-Standard nicht erwähnt. Ob sich da die Normierung über den Erfinder hinweggesetzt hat?)

Eine fiktive Liste mit Records des Typs *RecX*:

```
Rec1 (Startadresse 0008)
  Daten = 'Meier'
  Next = 0093    -> Rec2 (Startadresse 0093)
                      Daten = 'Müller'
                      Next = 0178     -> Rec3 (Startadresse 0178)
                                              ....
```

Die Zeigervariable im Feld *Next* enthält also jeweils die Startadresse des nächsten Listenelements[2]. Bezüge auf Records in einer derartigen Liste werden nach denselben Prinzipien notiert wie im zuvor gezeigten Beispiel mit *Screen*:

```
program List1;
type
  RecXPtr = ^RecX;
  RecX = record
           Daten: string[80];
           Next: RecXPtr;
         end;
var
  p,p1: RecXPtr;
  Rec1,Rec2: RecX;    { der Einfachheit halber statisch deklariert }
begin
  p := @Rec1;
  p^.Daten := 'Meier';            { setzt das Feld "Daten" von Rec1 }
  Rec1.Next := @Rec2;             { "Next" Rec1 auf die Adresse von Rec2 }
  Rec1.Next^.Daten := 'Müller';   { setzt das Feld "Daten" des Records,
                                    auf den Rec1.Next zeigt (d.h. Rec2) }

  Writeln(p^.Daten);          { ergibt 'Meier' }
  p1 := p^.Next;              { setzt p1 auf die Adresse von Rec2 }
  Writeln(p1^.Daten);         { ergibt 'Müller' }
  Writeln(p^.Next^.Daten);    { ergibt ebenfalls 'Müller' }
end.
```

Die Angabe *p^.Next^.Daten* wertet der Compiler offensichtlich von links nach rechts aus: zuerst wird *p* (und damit die Adresse von *Rec1* gelesen), danach das Feld *Next* innerhalb dieses Records (= Adresse von *Rec2*) und schließlich das Feld *Daten* dieses Records (= *Rec2.Daten*). Im Prinzip ließe sich die Reihe beliebig fort-

[1] z.B. in *Algorithmen und Datenstrukturen*, der Bibel zum Thema »Listen«.

[2] Die oft in diesem Zusammenhang verwendeten Termini »selbstbezüglich« und »rekursive Definition« sind faktisch Unsinn - schließlich zeigt der Inhalt des Feldes *Next* nicht wieder auf dieselben Daten, sondern lediglich auf eine weitere Variable desselben *Typs*.

setzen: *p^.Next^.Next^.Daten* ergibt den Inhalt von *Daten* eines Records, dessen Startadresse in *Rec2.Next* eingetragen ist usw. Praktische Anwendungen für derartige Monstren gibt es Wirth sei Dank nur wenige.

Explizite Typ-Umwandlungen

Etwas salopper als im Referenzhandbuch definiert: Bei den Formen *XTyp(Variable)* oder *XTyp(Ausdruck)* wendet der Compiler rücksichtslos die Eckdaten von *XTyp* an (Größe, Struktur und Typ der einzelnen Felder, soweit vorhanden); um den tatsächlichen Typ des »Klammerinhalts« kümmert er sich nicht sonderlich. Die häufigsten Anwendungen dieser Art von Umwandlung sind:

- Erzwingen bestimmter numerischer Formate zur Vermeidung von Überläufen. Ein typisches Beispiel:

```
var
  SectorCount, SecSize: Integer;  { Sektorzahl, Sektorgröße }
begin
  ...   { Ermittlung über DOS-Aufruf }
  Writeln(SectorCount * SecSize, ' Bytes Kapazität.');
  ...
```

  Der Compiler erwartet in diesem Fall ein Ergebnis vom Typ *Integer* - und *Writeln* gibt für alle Laufwerke mit mehr als 32 KByte Kapazität falsche Ergebnisse aus. Wenn wir dagegen auch nur einen der beiden Operanden explizit als *LongInt* darstellen, wird die Multiplikation auch in diesem Format ausgeführt und damit das richtige Ergebnis erzeugt:

```
Writeln(SectorCount * LongInt(SectorSize), ' Bytes Kapazität.');
```

  Vorsicht! Wer besonders gründlich sein will und gleich beide Operanden innerhalb *einer* Klammer als *LongInt* konvertiert, also

```
Writeln(LongInt(SectorCount * SectorSize), ' Bytes Kapazität.');
```

  - der wird sich wundern: Hier führt der Compiler zuerst die Multiplikation im Format *Integer* aus, erzeugt dabei einen Überlauf und versucht dann, das Ergebnis als *LongInt* zu behandeln.

- Konvertierungen generischer Zeiger »auf die Schnelle«. Funktionen wie *Ptr* liefern einen Zeigertyp zurück, dem der Compiler keine »Eckdaten« zuordnet (siehe vorletzter Abschnitt). Um beispielsweise eine einzelne Speicherzelle über *Ptr* zu lesen, kann das Funktionsergebnis entweder einem typisierten Zeiger zugeordnet werden, also

```
var BytePtr : ^Byte;    { typisierter Zeiger }
    VideoMode: Byte;    { momentan gesetzter Videomodus }
begin
  BytePtr := Ptr($40,$49);  { BytePtr auf die Adresse $0040:0049 }
  VideoMode := BytePtr^;    { "PEEK" }
end.
```

oder man spart sich den Umweg über die typisierte Zeigervariable und teilt dem Compiler direkt mit, daß das Ergebnis von *Ptr* als Zeiger auf ein einzelnes Byte behandelt werden soll:

```
var VideoMode: Byte;
begin
  VideoMode := Byte(Ptr($40,$49)^); { "das Byte, auf das der von Ptr
                                      erzeugte Zeiger zeigt" }
end.
```

- Konvertierungen untypisierter Parameter. Routinen wie *Move* oder *FillChar*, die Parameter als **var** *x;* deklarieren, bekommen bei ihrem Aufruf einen Zeiger für *x* übergeben (siehe Kapitel 1) - innerhalb der Routine muß dem Compiler erst mitgeteilt werden, wie er diesen Zeiger anwenden soll.
- Ermittlung von Wert- oder Variablen*teilen*, die ansonsten nicht oder nur unter größerem Aufwand erreichbar wären (siehe übernächster Abschnitt).

Zeiger- vs. Wertkonvertierungen und Probleme mit Arrays

Wie im Abschnitt »Variablengrößen und -strukturen« demonstriert, legt der Typ eines Zeigers fest, wieviele aufeinanderfolgende Bytes das Programm bei einem Zugriff liest oder schreibt. Mit einer expliziten Typ-Umwandlung läßt sich folgerichtig dasselbe erreichen:

```
var p: Pointer; x: LongInt;
 ...
 x:= Byte(p^);       { liest ein einzelnes Byte }
 x:= Integer(p^);    { liest zwei aufeinanderfolgende Bytes }
 x:= LongInt(p^);    { liest vier aufeinanderfolgende Bytes }
  ....
```

Bei einer Variablen ist die Anzahl der Bytes dagegen unveränderlich durch ihre Typ-Deklaration festgelegt - explizite Typ-Umwandlungen über die Grundtypen*) bestimmen hier nur, in welchem Format der Compiler den Wert *nach dem Lesen* interpretieren soll (z.B. für folgende Multiplikationen oder Zwischenspeicherungen):

*) Mit Records ist das erstaunlicherweise etwas anderes - siehe »Zugriffe auf Teile einer Struktur«.

```
var b: Byte; x: LongInt;
 ...
 x:= Byte(b);          { liest ein einzelnes Byte und behandelt den
                         gelesenen Wert als Byte }
 x:= Integer(b);       { liest ein einzelnes Byte und behandelt den
                         gelesenen Wert als Integer }
 x:= LongInt(b);       { liest ein einzelnes Byte usw. }
```

Das geht auch gar nicht anders - schließlich würde ein Befehl, der ab der Adresse von *b* zwei aufeinanderfolgende Bytes liest, im gezeigten Beispiel das niederwertige Byte von *x* »mitnehmen« (*x* ist hier direkt nach *b* im Datensegment gespeichert).

So weit, so gut. Unternehmen wir einen kurzen Ausflug auf den »Boot«-Sektor der Festplatte, dessen Kopfinformation in Peter Nortons *Programmer's Guide* folgendermaßen beschrieben ist:

```
Offset        Länge        Inhalt
  3           8 Bytes      ID (z.B. 'IBM 2.1')
 11           1 Wort       Sektorgröße in Bytes (z.B. $0200 = 512)
 13           1 Byte       Anzahl Sektoren/Cluster
 14           1 Wort       Anzahl reservierter Sektoren
 ...          ....
 19           1 Wort       Gesamtzahl Sektoren
 ...          ....
```

Wer eine derartige Struktur auswerten will, stößt auf eine ganze Reihe von Problemen:

Erster Versuch: Definition einer entsprechenden Record-Variablen. Funktioniert zumindest nicht direkt, weil die BIOS-Routinen immer einen ganzen Sektor lesen, der wesentlich mehr Daten als die gewünschte Kopfinformation enthält. (Was die Lösung über ein Puffer-Array und einen typisierten Zeiger angeht - siehe »Zugriffe auf Teile einer Struktur« auf den folgenden Seiten).

Zweiter Versuch: Definition eines Puffers als `array[0..511] of Byte`. Funktioniert - aber ausgesprochen mühselig. Zur Ermittlung der Gesamtkapazität ist beispielsweise die folgende Rechnung notwendig:

```
(Buffer[11] + 256 * Buffer[12])            { Sektorgröße in Bytes }
* (LongInt(Buffer[19] + 256 * Buffer[20])  { Gesamtzahl Sektoren }
   - Buffer[14] + 256 * Buffer[15])        { - reservierte Sektoren }
```

Dritter Versuch: Definition des Puffers als `array[0..255] of Word`, um sich die diversen Multiplikationen mit 256 zu sparen. Mit katastrophalem Ergebnis, weil das Programm nun für `Buffer[14]` nicht das 14. und 15. Byte des Arrays liest, *sondern das 14. Wort* - also die Bytes 28 und 29. Nach der Feststellung, daß ein schlichtes Halbieren der Indexe (*Buffer[7]* anstelle von *Buffer[14]*) auch nichts

nützt, weil einige Daten auf ungeraden Indexen anfangen, kehrt man normalerweise zum zweiten Lösungsansatz zurück.

Leider ist diese Problematik nicht auf Diskettensektoren beschränkt - wo auch immer man im Speicher des Computers herumsucht, stößt man auf derartige Datenstrukturen. Das folgende Programm verdeutlicht, welche Möglichkeiten eine explizite Typ-Umwandlung im Zusammenhang mit Arrays bietet (und welche nicht):

```
type
  IntPtrType = ^Integer;     { Zeiger auf Integerwerte }
var
  ByteArray: array[0..9] of Byte;
  IntPtr: IntPtrType;
  x: Integer;

begin
  for x:= 0 to 9 do
    ByteArray[x] := x;
          .
  Writeln(ByteArray[0]);              { ergibt 0 }
  Writeln(Integer(ByteArray[0]));     { dito - es wird nur ein Byte gelesen }

  Writeln(ByteArray[2] + 256 * ByteArray[3]);  { die "Methode zu Fuß" }

  IntPtr:= @ByteArray[2];             { IntPtr auf das Byte 02 des Arrays }
  Writeln(IntPtr^);                   { ** hier liest das Programm zwei
                                        aufeinanderfolgende Bytes! ** }

  Writeln(IntPtrType(@ByteArray[2])^); { dasselbe in direkter Form }
end.
```

Der letzte Aufruf von *Writeln* weist den Compiler an, die Adresse von *ByteArray[2]* als Adresse eines Integerwertes zu behandeln, und ist nur zur Demonstration vorhandener Möglichkeiten gedacht (derartig undurchsichtige Operationen halte ich für »schlechten Programmierstil«).

Zugriffe auf Teile einer Struktur

Wie im letzten Abschnitt durch eine verschämte Fußnote angedeutet, unterscheidet Pascal eigentlich nicht zwei, sondern drei Anwendungen der expliziten Typ-Umwandlung:

- Konvertierung der »Eckdaten« von Zeigern (*XTyp(p^)*);
- Interpretation eines zuvor gelesenen Wertes (*XTyp(Variable)*);
- und die Umdeklarierung eines Zugriffs über Record-Definitionen, die auch bei fix deklarierten Variablen funktioniert. Eine formale Entschuldigung für diese

eigentlich unlogische Möglichkeit könnte sein, daß dabei in jedem Fall mit Adressen gearbeitet wird.

Die Konvertierung über Zeiger-Eckdaten

Der Compiler analysiert nicht, welchen Wert eine Variable zu einem bestimmten Zeitpunkt haben könnte, und behandelt voneinander getrennte Anweisungen separat. Dieser Grundsatz gilt auch für Zeigervariablen - der expliziten Typ-Konvertierung eines generischen Zeigers sind *deshalb* praktisch keine Grenzen gesetzt, weil der Compiler gar nicht wissen will, worauf diese Variable momentan zeigt. Die programmiertechnisch sauberste Lösung des zuvor gezeigten »Boot-Record-Zugriffproblems« besteht aus drei Schritten:

```
type   { 1. Typdefinition }
  BootRec = record
 { Offset: } Fill:         array[0..2] of Byte;  { drei Bytes }
   {  3 }    IBM_ID:       array[0..7] of Char;  { 'IBM 2.1...' }
   { 11 }    Bytes_Sec:    Word;  { Bytes pro Sektor }
   { 13 }    Secs_Cluster: Byte;  { Sektoren pro Cluster }
   { 14 }    ....

var    { 2. Deklaration eines *generischen* Zeigers }
  p: Pointer;
  Buffer: array[0..511] of Byte; { Puffervariable }
  ....
begin
   .... { Einlesen des "Boot"-Sektors in das Array Buffer }

   p:= @Buffer;        { 3. Setzen von p auf den Puffer und }
   with BootRec(p^) do      { Typ-Konvertierung von p }
   begin
     Writeln('Bytes pro Sektor: ', Bytes_Sec);       { = p^.Bytes_Sec }
     Writeln('Sektoren/Cluster: ', Secs_Cluster);    { = p^.Secs_Cluster }
   ....
```

Die Formulierung `BootRec(p^)` läßt sich wohl am besten (und wieder nicht Referenzhandbuch-kompatibel) so übersetzen: »tu so, als ob *p* auf einen Record des Typs *BootRec* zeigen würde«.

Die Umdeklarierung von Variablen über Recordtypen...

läuft nach demselben Prinzip, kommt aber (wie gesagt: erstaunlicherweise) ohne den Umweg über Zeiger aus. Da der Compiler in diesem Fall sehr wohl weiß, welchen Typ die Variable im Normalbetrieb hat, macht er eine zusätzliche Einschränkung: Die Größe der Variablen muß mit der des »darüberdefinierten« Records exakt übereinstimmen. Wie in der Datei README*) kurz erläutert, läßt sich ein

*) Gemeint ist hier die Version 4.0. In der Version 5.0 findet sich das entsprechende Material in Kapitel 15.

LongInt mit der folgenden Definition in eine untere und eine obere Worthälfte zerlegen:

```
type
  TwoWords = record  { zum Zerlegen von LongInts }
     LoWord: Word;     { niederwertiges Wort }
     HiWord: Word;     { höherwertiges Wort }
   end;

var   X: LongInt;
      A,B: Word;

begin
  X := $1010AABB;
  A:= TwoWords(X).LoWord;   { = $AABB }
  B:= TwoWords(X).HiWord;   { = $1010 }
end.
```

Die Übersetzung der drei Zuweisungen durch den Compiler:

```
** X := $1010AABB;
0007 C7060000BBAA   MOV    Word Ptr [X],AABB      ; niederwertiges Wort
000D C70602001010   MOV    Word Ptr [X+02],1010   ; höherwertiges Wort
** A:= TwoWords(X).LoWord;
0013 A10000         MOV    AX,[X]                 ; obere Hälfte wird ignoriert
0016 A30400         MOV    [A],AX                 ; unteres Wort -> Variable A
** B:= TwoWords(X).HiWord;
0019 A10200         MOV    AX,[X+02]              ; nur die obere Hälfte lesen
001C A30600         MOV    [B],AX                 ; -> Variable B
```

Hier wird also tatsächlich nicht mit irgendwelchen Maskierungen (wie AND $0000FFFF) gearbeitet, sondern direkt auf den entsprechenden Teil der Variablen *x* zugegriffen.

Zeigerzerlegungen

Zeiger und Adressen werden wie *LongInts* mit vier Bytes gespeichert und lassen sich auf dieselbe Weise in zwei *Word*-Hälften teilen - nur daß wir hier nicht einen höherwertigen und einen niederwertigen Teil erhalten, sondern eben *Segment* und *Offset* (vgl. Anhang A):

```
type
  PWord = record   { zum Zerlegen von Zeigern und Adressen }
            Off: Word;  { Offset-Anteil }
            Seg: Word;  { Segment }
          end;

var
  s: string;
  p: Pointer;
```

```
begin
  p := @s;

  Writeln('Adresse von s: ', Seg(s),      ':', Ofs(s));
  Writeln('p zeigt auf s: ', PWord(p).Seg, ':', PWord(p).Off);

  Writeln('Adresse von p: ', Seg(p),      ':', Ofs(p));
end.
```

Der letzte Aufruf von *Writeln* in diesem Programm demonstriert, wieso wir für das »Splitting« eines Zeigerwertes ohne eine explizite Typ-Umwandlung nicht auskommen: *Seg(p)* und *Ofs(p)* liefern die Adresse zurück, *auf der* p *gespeichert ist* - und nicht den Inhalt dieser Variablen.
Auf *Konstanten* läßt sich diese Art von Konvertierung übrigens nicht anwenden - eine Formulierung wie

```
Writeln(PWord(@x).Seg, ':', PWord(@x).Off);
```

wird vom Compiler mit der Meldung »Invalid type cast« zurückgewiesen.

Was den Zweck derartiger Tricksereien betrifft: über die Deklaration von *PWord* können wir einer Zeigervariablen recht einfach

- ein anderes Segment bei gegebenem Offset unterschieben, wie es beim Entfernen speicherresidenter Programme notwendig ist (siehe Kapitel 4).
- Segment- und Offset-Anteile von Zeigern wie normale *Word*-Variablen manipulieren, was einem in manchen Fällen viele Typ-Deklarationen erspart. Nach der Definition des Typs *PWord* läßt sich der Wert einer Zeigervariablen *p* beispielsweise mit einem schlichten

  ```
  Inc(PWord(p).Off, x)
  ```

  um *x* erhöhen - eine Operation, die ansonsten die Deklaration eines Zeigers auf ein *Byte*-Array erfordern würde (und dazu noch die Abschaltung der Bereichsüberprüfung, wenn *p* über die Grenzen des »Arrays« hinausbewegt werden soll).

Ungewöhnliche Typ-Umwandlungen

Das folgende Programm ist sinnlos, zwecklos, enthält nichts Neues und erst recht keine Kommentare - aber das Ergebnis ist lustig:

```
type FC = record
            d,c,b,a: Char;
          end;
var  x: LongInt;
begin
  x := 1098211617;
  with FC(x) do Writeln(a,b,c,d);
end.
```

»Statische« Anwendungen

Zeiger scheinen auf irgendeine rätselhafte Weise an die Papiersorte gebunden zu sein, auf der Listings abgedruckt werden: In den bekannteren Computerzeitschriften erscheinen sie fast nur im Zusammenhang mit Utilities (»wenn man hier bohrt und da dreht, dann macht der Computer...«) - in akademischen Lehrbüchern verwandeln sie sich dagegen augenblicklich in Bestandteile entsetzlich komplizierter Sortier- und Organisationsverfahren. (Auch Professor Wirth in persona macht da keine Ausnahme).

Verzichten wir also für einen Moment auf beide Extreme und wenden wir uns einem wirklich alltäglichen Problem zu - dem Sortieren einer begrenzten Anzahl von Strings.

StrSort1 - die Standardvariante

StrSort1 deklariert ein statisches Stringarray und besteht aus drei Teilen:

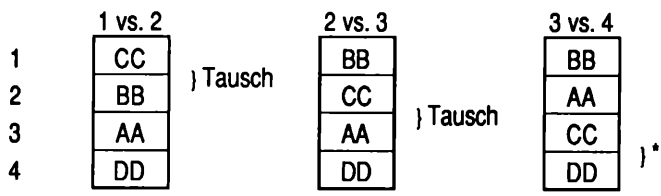
- Sämtliche Elemente des Arrays werden über eine »Zufallsfunktion« (*RandomLine*) gesetzt,
- danach mit dem einfachsten aller Sortieralgorithmen (Bubblesort) in alphabetische Reihenfolge gebracht
- und schließlich ausgegeben.

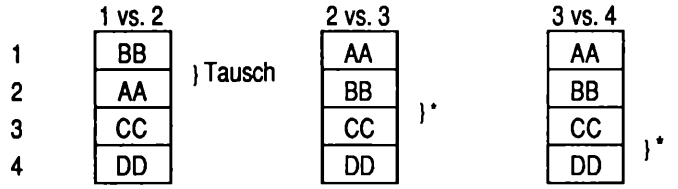
Natürlich gibt es wesentlich effizientere Sortiermethoden als Bubblesort, außerdem sortiert das Programm streng nach den ASCII-Codes ('A'..'Z' zuerst, danach 'a'..'z'), Umlaute sind von vornherein ausgeklammert usw. - aber hier geht es schließlich um eine Demonstration.

```
program StrSort1;

const
  NumLines = 200;      { Anzahl der Strings }
  MaxLineLen = 80;     { Maximallänge eines Strings }

type
  LineStr = string[MaxLineLen];

var
  Lines: array[1..NumLines] of LineStr;
  TempLine: LineStr;  { für das Vertauschen zweier Strings }
  x: Integer;
  Done: Boolean;
```

```
function RandomLine: LineStr;
{ Liefert einen "zufälligen" String zurück }
var Len: Integer;
    s: LineStr;
begin
  Len := Random(MaxLineLen);  { zufällige Länge }
  s[0] := chr(Len);           { wird direkt gesetzt }
  FillChar(s[1],Len,          { Füllen des Strings }
        Chr(Random(ord('z')-31)+32));  { mit einem durchgehenden Zeichen }
  RandomLine := s;
end;

begin
          { <== KEIN Randomize, damit wir immer dieselben Strings erzeugen }

  for x:= 1 to NumLines do    { 1. "Zufällige" Initialisierung }
    Lines[x] := RandomLine;

  repeat                      { 2. Bubblesort }
    Done := True;
    for x:= 1 to NumLines-1 do
      if Lines[x] > Lines[x+1] then
      begin
        TempLine := Lines[x];     { Austausch }
        Lines[x] := Lines[x+1];
        Lines[x+1] := TempLine;
        Done := False;
      end;
  until Done;

  for x:= 1 to NumLines do    { 3. Ausgabe des sortierten Arrays }
    Writeln(Lines[x]);
end.
```

Das Programm vergleicht jeweils zwei aufeinanderfolgende Array-Elemente: wenn das erste größer als das zweite ist, werden die beiden ausgetauscht. Ein Beispiel für zwei Durchläufe:





Das Element *AA* steigt hier wie eine Seifenblase (»Bubble«) zum Anfang des Arrays - jeder Durchlauf der mit *repeat..until* aufgebauten Schleife bewegt es um eine weitere Position nach vorne.

Wenn wir davon ausgehen, daß die 200 Elemente des Stringarrays *Lines* durchschnittlich 40 Zeichen lang sind und sich das Element *AA* vor dem Sortieren in der Mitte des Arrays befindet, dann schaufelt das Programm also mindestens 8000 Bytes durch den Speicher, bis *AA* an seiner endgültigen Position angekommen ist. Dementsprechend ist der Zeitbedarf des Programms: er liegt auf einem AT mit 12 MHz bei 2,85 Sekunden.

StrSort2 - dasselbe Programm mit Zeigern

Die zweite Version von *StrSort* unterscheidet sich von der ersten nur dadurch, daß sie die Stringvariablen nicht statisch, sondern als Zeiger-Array deklariert. Ein Umkopieren der String*inhalte* ist hier unnötig - stattdessen werden einfach die entsprechenden Zeigervariablen vertauscht. Gegenüber *StrSort1* hat das Programm zwei gewaltige Vorteile:

- der Gesamtumfang der Strings ist nicht mehr auf 64 KByte begrenzt (vgl. Benutzerhandbuch, Kapitel 6);
- weil bei jedem Vertauschen zweier Elemente nur noch jeweils acht (anstelle von durchschnittlich 80) Bytes umkopiert werden müssen, reduziert sich die Ausführungszeit fast auf die Hälfte (1.65 anstelle von 2.85 Sekunden).

```
program StrSort2;

const
  NumLines = 200;
  MaxLineLen = 80;

type
  LineStr = string[MaxLineLen];
```

```
var
  LinePtrs: array[1..NumLines] of ^LineStr;  { Stringzeiger-Array }
  TempPtr: Pointer;
  x: Integer;
  Done: Boolean;

function RandomLine: LineStr;
{ Liefert einen "zufälligen" String zurück }
var Len: Integer;
    s: LineStr;
begin
  Len := Random(MaxLineLen);  { zufällige Länge }
  s[0] := chr(Len);           { wird direkt gesetzt }
  FillChar(s[1],Len,          { Füllen des Strings }
        Chr(Random(ord('z')-31)+32));  { mit einem durchgehenden Zeichen }
  RandomLine := s;
end;

begin
          { <== KEIN Randomize, damit wir immer dieselben Strings erzeugen }
  for x:= 1 to NumLines do
  begin
    New(LinePtrs[x]);
    LinePtrs[x]^ := RandomLine;
  end;

  repeat                      { 2. Bubblesort }
    Done := True;
    for x:= 1 to NumLines-1 do
      if LinePtrs[x]^ > LinePtrs[x+1]^ then
      begin
        TempPtr := LinePtrs[x];       { Austausch der Zeiger - und }
        LinePtrs[x] := LinePtrs[x+1]; { nicht der Stringinhalte }
        LinePtrs[x+1] := TempPtr;
        Done := False;
      end;
  until Done;

  for x:= 1 to NumLines do     { 3. Ausgabe des sortierten Arrays }
    Writeln(LinePtrs[x]^);
end.
```

»Dynamische« Variablen

Die vorangehenden Abschnitte dieses Kapitels dürften in ihrer Art einen Rekord darstellen: 20 Seiten über Zeiger - und nirgendwo das Wort »dynamisch«. Dafür läßt sich dieses Thema aber jetzt auch auf knapp zwei Seiten komplett abhandeln:

1. Beim Start eines Programms wird eine Zeigervariable (*HeapPtr*) auf die Adresse der ersten freien Speicherstelle hinter dem Programmende gesetzt (sozusagen mit *HeapPtr* := @<Programmende + 1>). Die weiteren - und hier unwichtigen - Details (Größe des Heaps, Adresse der Fragmentliste usw.) sind in Kapitel 1 beschrieben.

2. Die Prozedur *GetMem* erwartet eine untypisierte (oder beliebig typisierte) Zeigervariable *p* als **var**-Parameter und dazu eine Größenangabe *x*. Die übergebene Zeigervariable bekommt den momentanen Wert von *HeapPtr* zugewiesen, danach wird *HeapPtr* um *x* Bytes erhöht*). Die *x* Speicherzellen »zwischen« *p*^ und der Adresse, auf die *HeapPtr* nach dieser Erhöhung zeigt, gelten als »reserviert«. Ein Beispiel:

```
var  p1,p2,p3: Pointer;
 ...
 { Annahme: HeapPtr zeigt momentan auf die Adresse "1000" }
 GetMem(p1,100);   { setzt p1 auf "1000", HeapPtr auf "1100" }
 GetMem(p2,50);    { setzt p2 auf "1100", HeapPtr auf "1150" }
 GetMem(p3,300);   { setzt p3 auf "1150", HeapPtr auf "1450" }

 GetMem(p1,150);   { setzt p1 auf "1450", HeapPtr auf "1600"
                     und hinterläßt ein permanentes Loch im
                     Bereich "1000".."1099" }
```

3. Die Prozedur *New* erwartet eine typisierte Zeigervariable *p* als **var**-Parameter. Der Compiler setzt hier selbständig den Umfang des »Zielvariablentyps« ein (z.B. 4, wenn *p* als *^LongInt* deklariert ist) und ruft dann *GetMem* auf. Dieses automatische Einsetzen der Größe ist der einzige Unterschied zwischen *New* und *GetMem* - eine Routine mit dem Namen *New* gibt es in der Laufzeitbibliothek nicht.

4. Die Prozedur *FreeMem* erwartet eine untypisierte Zeigervariable *p* als **Wert**-Parameter und eine Größenangabe *x*, wobei *x* denselben Wert wie bei der Belegung des Speicherbereichs haben muß. Sie gibt den Bereich von *x* Bytes wieder frei, dessen Startadresse in *p* enthalten ist. Diese »Freigabe« geschieht entweder durch ein Herabsetzen von *HeapPtr* um *x* oder durch einen Eintrag in die Fragmentliste (siehe Referenzhandbuch, Kapitel 25). Der Wert von *p* und der Inhalt des »freigegebenen Speicherbereichs« werden durch *FreeMem* nicht verändert:

*) Die Fragmentliste ist hier nicht berücksichtigt - was der Prinzipbeschreibung aber keinen Abbruch tun sollte.

```
var
  IntPtr: ^Integer;
  StrPtr: ^string;

begin
  New(IntPtr);          { erhöht HeapPtr um 2 }
  IntPtr^:= 15;         { schreibt den Wert 15 in den reservierten Bereich }
  FreeMem(IntPtr,2);    { erniedrigt HeapPtr wieder um 2 }
  Writeln(IntPtr^);     { ergibt 15 - unverändert }

  New(StrPtr);            { erhöht HeapPtr um 256 }
  Writeln(IntPtr^);       { ergibt immer noch 15 }
  StrPtr^ := 'Hallo!';    { **jetzt** wird der zuvor freigegebene
                            Speicherbereich (IntPtr^) überschrieben }
  Writeln(StrPtr^);     { ergibt 'Hallo!' }
  Writeln(IntPtr^);     { ergibt irgendeinen Unsinn }
end.
```

• Eine Zeigervariable muß nach der »Freigabe« ihres Bereichs deshalb als »undefiniert« betrachtet werden, weil *folgende* Aufrufe von *New* oder *GetMem* diesen Bereich neu belegen (d.h. eine andere Zeigervariable darauf setzen) können.

• Damit ein Bereich freigegeben werden kann, muß eine Zeigervariable existieren, die die Startadresse dieses Bereichs enthält (d.h. »auf diesen Bereich zeigt«). Zwei aufeinanderfolgende Belegungen über *New* oder *GetMem* mit *derselben* Zeigervariablen verändern diese Variable zweimal - wenn ihr Wert nicht zwischenzeitlich woanders gespeichert wurde, läßt sich der zuerst belegte Bereich bis zum Ende des Programms überhaupt nicht mehr freigeben.

5. *Dispose* erwartet einen typisierten Zeiger (ebenfalls als Wertparameter). Der Compiler setzt die Größe der »Zielvariablen« ein und erzeugt dann einen Aufruf von *FreeMem*. (Ergo: eine Routine namens *Dispose* sucht man in der Laufzeitbibliothek ebenfalls vergebens).

6. *Mark(p)* besteht (hauptsächlich) aus der Zuweisung *p := HeapPtr*, *Release(p)* aus der Zuweisung *HeapPtr := p*. Wenn wir den Stand von *HeapPtr* bei "1100" mit *Mark* in einer Variablen *p* speichern und zu irgendeinem späteren Zeitpunkt *Release* mit dieser Variablen aufrufen, dann wird *HeapPtr* wieder auf "1100" gesetzt: Der nächste Aufruf von *New* oder *GetMem* fängt dann wieder mit "1100" an.

Was den gesamten Rest (Zugriffe, Typ-Deklarationen, Umwandlungen usw.) betrifft: Es spielt überhaupt keine Rolle, ob ein Zeiger auf eine statische Variable (im Datensegment des Programms), auf irgendeine andere Adresse (z.B. den Bildspeicher) oder auf den Heap zeigt.

Ein Nachgedanke zur Definition des Begriffs an sich: Eine Zeigervariable, die mit

```
var   p: Pointer;
```

deklariert wurde, ist *statisch*[*] - schließlich hat der Compiler einen festen Speicherplatz dafür reserviert. Ob sie nun zufällig auf einen Bereich innerhalb des Heaps zeigt oder nicht, sollte dabei eigentlich keine Rolle spielen: *Integer*-Variablen klassifiziert man ja auch nicht nach ihrem momentanen Wert. Ein Aufruf wie

```
GetMem(p,300);
```

erzeugt dagegen eine dynamische Variable - nämlich den »namenlosen« Speicherbereich, auf dessen Startadresse *p* dadurch gesetzt wird.

Zeiger auf Pascal-Routinen in der Version 4.0

Die Funktionen *Addr*, *Seg* und *Ofs* lassen sich in Turbo Pascal (genauso wie der Adress-Operator @) nicht nur auf Variablen, sondern auch auf Prozeduren und Funktionen anwenden. Zwei Einschränkungen sind in diesem Zusammenhang allerdings zu berücksichtigen - und die sind manchmal höchst ärgerlich:

- Zeiger auf Pascal-Routinen müssen untypisiert sein. Weder Turbo Pascal noch der ANSI-Standard definieren ein Analogon zum C-Konstrukt »Zeiger auf Routine mit dem Ergebnistyp xxx«;
- Der »Zugriff« über einen Zeiger ist in Turbo Pascal bei Routinen nur auf dieselbe Weise möglich wie bei Daten: *Aufrufe* über Zeiger - die berühmten »prozeduralen Parameter« - sind nicht im Sprachumfang enthalten.

Am ersten Punkt läßt sich wohl ohne einen Umbau des Compilers nicht allzuviel ändern - vorausgesetzt, daß wir mit dem zweiten Punkt zu Rande kommen, liegt die Verantwortung für übereinstimmende Parameterdeklarationen und Ergebnistypen beim Programmierer.

Was diesen zweiten Punkt betrifft, macht Turbo Pascal offensichtlich eine ganze Reihe »interner« Ausnahmen - von Exit-Prozeduren über *ExitProc* und der Heap-Fehlerbehandlung über *HeapError* bis hin zu **interrupt**-Prozeduren, die über Interruptvektoren des Systems aufgerufen werden können (siehe Kapitel 4).

Für *diese* Zwecke wurde die Möglichkeit in den Compiler eingebaut, Adress-Ermittlungen und Zeiger auch auf Routinen anzuwenden - das Beispielprogramm PROCPTR.PAS (auf der Diskette II) scheint dagegen eher ein Nachgedanke der Entwickler gewesen zu sein.

Die folgenden Abschnitte nehmen die Idee von PROCPTR.PAS auf und bauen diesen Nachgedanken kräftig aus, wobei wir größtenteils auf das Material von Kapitel 1 zurückgreifen.

[*] Jedenfalls meiner Auffassung nach.

PROCFUNCPTR.PAS

Die dem Demonstrationsprogramm PROCPTR.PAS zugrundeliegende Idee, eine Zeigervariable auf eine Routine zu setzen und diese Routine dann mit einem **inline**-Befehl aufzurufen, läßt sich ohne jede Einschränkung auch auf Funktionen übertragen. Wir brauchen lediglich eine weitere **inline**-Routine dazu, die mit dem entsprechenden Ergebnistyp als *Funktion* deklariert ist*):

```
program ProcFuncPtr;

var
  ProcAddr, FuncAddr : Pointer;  { Startadressen }
  i: Integer;

procedure CallFarProc(var i: Integer; w: Word; s: string);
  inline($FF/$1E/ProcAddr);   { CALL FAR [ProcAddr] }

function CallFarFunc(var i :Integer; w :Word; s: string): Integer;
  inline($FF/$1E/FuncAddr);   { CALL FAR [FuncAddr] }

{$F+}   { *** unbedingt notwendig! *** }
procedure Proc1(var i: Integer; w: Word; s: string);
begin
  Write('Proc1: ');
  Writeln('i = ', i,'  w = ', w, '  s = ', s);
end; {$F-}

{$F+}   { *** unbedingt notwendig! *** }
function Func1(var i : Integer; w : Word; s : string): Integer;
begin
  Write('Func1: ');
  Write('i = ', i,'  w = ', w, '  s = ', s);
  Func1 := 77;
end; {$F-}

begin
  i := 20;
  ProcAddr := @Proc1;  { ProcAddr auf Proc1 }
  CallFarProc(i, 7, 'irgendein Text');

  FuncAddr := @Func1;  { FuncAddr auf Func1 }
  Writeln(CallFarFunc(i, 5, 'noch mehr Text...'));
end.
```

*) Die Formate und Reihenfolge von Parameterübergaben, ihre Lage auf dem Stack, der Aufruf von inline-Routinen, die notwendigen Vorsichtsmaßnahmen usw. usf. machen ungefähr 30 Seiten des Kapitels 1 aus - ich habe nach einem kurzen Versuch die Halbheiten bleiben lassen und bitte Sie, die entsprechenden Abschnitte nachzuschlagen (soweit notwendig). Oder auch: Prozeduraufrufe über Zeiger gehören nun einmal in das Kapitel über Zeiger, drum stehen sie drin.

Nachdem man das Ergebnis dieses Programms, also

```
Proc1: i = 20  w = 7  s = irgendein Text
Func1: i = 20  w = 5  s = noch mehr Text...77
```

ausgiebig bewundert hat, stellt sich dann doch über kurz oder lang die Frage, was man damit anfangen kann - schließlich ist ein Aufruf wie

```
Proc1(i,7,'irgendein Text');
```

anstelle der gezeigten Methode nicht nur wesentlich übersichtlicher, sondern macht auch weniger Tipparbeit. Das Prinzip läßt sich aber erweitern:

PROCARRAY.PAS - Aufruf über Funktionsnummern

Aus der Programmiersprache C ist ein Konstrukt bekannt, dessen Pascal-Äquivalent ungefähr so aussieht:

```
if FCode >= 0 and FCode <= 5 then CallFarProc(FCode)
  else Write('Unbekannter Befehl!');
```

Übersichtlicher ist so etwas nicht gerade, weil man erst einmal genau prüfen muß, wohin der Aufruf mit einer bestimmten Funktionsnummer führt - zum einen erspart es aber die ansonsten übliche und ermüdende **case**-Anweisung:

```
case FCode of
  1 : Proc1;
  2 : Proc2;
  3 : Proc3;
  4 : Proc4;
  5 : Proc5;
 else
  Writeln('Unbekannter Befehl!');
end; { case }
```

Zum anderen: Gegenüber der Herumspringerei, die der Prozessor in einer **case**-Anweisung ausführen muß, ist ein solcher Aufruf geradezu teuflisch schnell.

Die praktische Ausführung der Idee:

```
program ProcPtrArray;

const
  Num1  = 1;

var
  FPP: array[Num1..3] of Pointer;  { Startadressen }
  x: Integer;

{$F+} procedure Proc1(i: Integer);  {$F-}
begin   Writeln('Erste Prozedur: ', i);  end;
```

```
{$F+} procedure Proc2(i: Integer);  {$F-}
begin   Writeln('Zweite Prozedur: ', i); end;

{$F+} procedure Proc3(i: Integer); {$F-}
begin   Writeln('Dritte Prozedur: ', i); end;

procedure CallFarProcArray(i: Integer; ProcNum: Integer);
{ Holt den Parameter ProcNum vom Stack, läßt alle anderen Parameter
  drauf und ruft die Prozedur auf, deren Adresse in FFP[ProcNum]
  gespeichert ist. }
inline ($5B/            { POP  BX        ; ProcNum vom Stack }
        $83/$EB/<Num1/  { SUB  BX,<Num1  ; minus Array-Start }
        $D1/$E3/        { SHL  BX,1      ; * 2               }
        $D1/$E3/        { SHL  BX,1      ; -> * 4            }
        $81/$C3/FPP/    { ADD  BX,FPP    ; + Start des Arrays}
        $FF/$1F);       { CALL FAR[BX]   ; Aufruf            }

begin
  { "Initialisierung" des Startadressen-Arrays }
  FPP[1] := @Proc1; FPP[2] := @Proc2; FPP[3] := @Proc3;

  for x:= 1 to 3 do
    CallFarProcArray(x+100, x);
end.
```

Einige Bemerkungen und Warnungen dazu (deren Hintergründe sämtlich in Kapitel 1 besprochen sind):

- Alle über *CallFarProcArray* aufgerufenen Routinen müssen dieselben Parameter deklarieren. Weil der eigentliche Aufruf hier über eine Folge direkt eingefügter Prozessorbefehle geschieht, hat der Compiler *keine Prüfmöglichkeit.*
- Die »Nummer« der aufzurufenden Prozedur muß (unabhängig davon, wieweit Sie *CallFarProcArray* erweitern) *immer der letzte Parameter* sein - falls nicht, holt der Befehl POP BX rücksichtslos irgendeinen anderen Wert vom Stack und benutzt ihn zur Indizierung von *FPP*.
- Eine Version von *CallFarProcArray*, die Funktionen aufruft, erfordert genau dieselben Modifikationen wie *CallFarFunc* im Programm PROCFUNCPTR: Deklarieren Sie die Routine als Funktion, legen Sie einen Ergebnistyp fest und lassen Sie den eigentlichen **inline**-Code unverändert.
- **Alle Routinen, die über Zeiger aufgerufen werden, müssen als *far* deklariert sein.** Der Aufruf von *near*-Funktionen und Prozeduren über ein Array ist eine fast unmögliche Angelegenheit, weil der Prozessor bei dieser Art von Aufrufen mit relativer Adressierung arbeitet (vgl. »Relozierung« in Kapitel 1).

Version 5.0

Offensichtlich haben die Entwickler von Borland das Flehen der Benutzer erhört*) - und dem Compiler in dieser Version einen prozeduralen Variablentyp spendiert, der die zuvor besprochenen **inline**-Tricksereien unnötig macht. (Falls Sie mit der Version 4.0 arbeiten und diesen Abschnitt nur als Ausblick auf zukünftige Anschaffungen lesen: Im Prinzip gelten für prozedurale Variablen dieselben Einschränkungen wie für **inline**-Konstrukte (äquivalente Parametertypen und -reihenfolgen, Codierung als *far* usw.) - mit dem Unterschied, daß der Compiler derartige Deklarationen auf Regelverletzungen hin prüfen kann).

Kurz und gut - was auf den vorangegangen Seiten als Material für absolute Spezialisten vorgestellt wurde, gehört in der Version 5.0 standardmäßig zum Sprachumfang.

Um 4.0-Programmierern eine Vergleichsmöglichkeit zu geben, 5.0-Programmierern, die nicht mit der deutschen Ausgabe des Handbuchs arbeiten, ein Anwendungsbeispiel zu bescheren, und nicht zuletzt zur Dokumentation der Tatsache, daß auch dieses Kapitel überarbeitet wurde, sei hier das Listing des 5.0-Gegenstücks zu PROCARRAY wiederzugeben:

```
program ProcPtrArray50;

type
  IntProc = procedure(i: Integer);

var
  ProcPointers: array[1..3] of IntProc;
  x: Integer;
```

Nach der Definition von *IntProc* vereinbart die Deklaration von *ProcPointers* letztlich nichts weiter als ein Array, dessen Elemente aus Variablen des Typs *Pointer* bestehen, jeweils 4 Bytes Speicherplatz belegen und eine (vollständige) Adresse speichern können.

Der Compiler behandelt diese Variablen allerdings auf recht spezielle Weise: er setzt sie nicht mit Zeigern auf Routinen des Typs *IntProc*, sondern mit *Routinen dieses Typs* gleich. (Was letztlich eigentlich einen Widerspruch zur Sprachdefinition darstellt, bei Aufrufen über diese Zeiger aber dieselbe Syntax wie bei »direkten« Aufrufen ergibt - und darauf kam es den Entwicklern offensichtlich an).

```
{$F+}   { <- ALLE via Zeiger aufgerufenen Routinen als FAR! }
procedure Proc1(x: Integer);
begin   Writeln('Erste Prozedur: ', x);  end;

procedure Proc2(y: Integer);
begin   Writeln('Zweite Prozedur: ', y); end;
```

*) Im Gegensatz zur Dokumentationsabteilung - das Kapitel 6 des deutschen 5.0-Benutzerhandbuchs wird man in der amerikanischen Originalausgabe vergeblich suchen.

```
procedure Proc3(y: Integer);
begin   Writeln('Dritte Prozedur: ', y); end;
{$F-}
```

Die drei Prozeduren dieses Beispielprogramms folgen denselben Regeln wie in PROCARRAY: Sie müssen sämtlich als *far* codiert sein und Parameter gleichen Typs sowie gleicher Reihenfolge deklarieren.

```
begin
  ProcPointers[1] := Proc1;
  ProcPointers[2] := Proc2;  { <- kein '@' vor dem Prozedurnamen! }
  ProcPointers[3] := Proc3;

  for x:= 1 to 3 do
    ProcPointers[x](x);      { Aufrufe }
end.
```

Die Initialisierung des Array *ProcPointers* muß hier (wie in PROCARRAY) zur Laufzeit des Programms geschehen - Deklarationen prozeduraler Variablen als typisierte Konstanten werden vom Compiler als unzulässig moniert*).

Der Aufruf der drei Prozeduren durch das Hauptprogramm findet hier direkt statt. Die Übergabe einer Routinennummer an einen »Dispatcher« wie *CallFarProcArray* ist hier nicht mehr notwendig - genausowenig wie die diversen »Vorsichts« des letzten Abschnitts: Fehlende oder überzählige Parameter, unzulässige Typen, versehentliche *near*-Codierungen usw. werden automatisch entdeckt.

Auch wenn er etwas ungewöhnlich aussieht, hat der Aufruf über *ProcPointers* tatsächlich dieselbe Syntax wie der Aufruf einer normalen Prozedur. In einem »arraylosen« Beispiel läßt sich diese Behauptung etwas anschaulicher darstellen: Nach der Deklaration einer Variablen des Typs *IntProc* und ihrer Zuordnung zu einer Prozedur unseres Beispielprogramms, also

```
var OneProc: IntProc;
  ...
  OneProc := Proc2;     { <- auch hier: kein '@' vor dem Prozedurnamen! }
```

läßt sich durch einen (isolierten) Blick auf das Listing nicht mehr feststellen, ob *OneProc* eine echte Routine oder eben eine prozedurale Variable ist:

```
OneProc(z);             { Aufruf von Proc2, z = beliebiger Integer }
```

In einem Punkt werden die Unterschiede zwischen Zeigern und prozeduralen Variablen übrigens recht deutlich: Jeder Versuch einer expliziten Typ-Umwandlung - wie beispielsweise

```
type  ProcType = procedure;
      IntProcType = procedure(I: Integer);
```

*) Dasselbe gilt für den Variablentyp *Pointer* und müßte meines Wissens eigentlich nicht sein - schließlich werden die Adressen durch den Linker eingesetzt.

```
var   ProcPtr : ProcType;
  ...
  ProcPtr := IntProcType(Proc2);
```

stößt auf den energischen Widerstand des Compilers. (Was in der Sprache »C« nicht der Fall ist - offensichtlich haben die Turbo Pascal-Entwickler gewissen Wert auf einen letzten Rest von Formalismus gelegt).

Rekursion

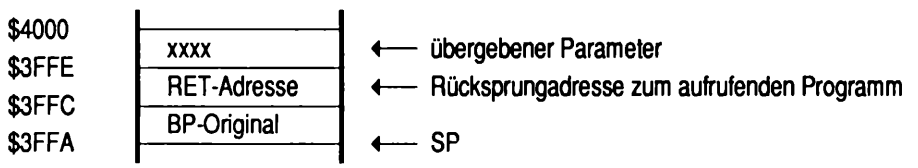
Wie in der Einleitung des Kapitels angedeutet, stellt dieses Thema jeden Hand- oder Lehrbuchschreiber vor ähnliche Schwierigkeiten wie Zeigervariablen - eigentlich sind die Hürden hier sogar noch ein Stückchen höher. Eine solide Erläuterung fordert

- vom Leser das Verständnis des Stackkonzepts und lokaler Variablen, sowie
- vom Autor ein brauchbares Anwendungsbeispiel. Weil anscheinend jeder bei Rekursion *sofort* an dynamische Listenstrukturen denken muß (die zu allem Überfluß auch noch Zeigervariablen voraussetzen und deren Elemente sich bei Programmierfehlern unauffindbar im Speicher zu tummeln pflegen), begnügt man sich fast immer mit *n*-Fakultät - einem »Problem«, das sich über eine Schleife wesentlich einfacher lösen läßt.

Dabei liefert DOS mit seinen hierarchischen Directories eine ideale Demonstrationsmöglichkeit für rekursive Programmierung sozusagen frei Haus - mit dynamischen Listen schlägt sich deshalb erst der dritte Teil dieses Kapitels herum*).

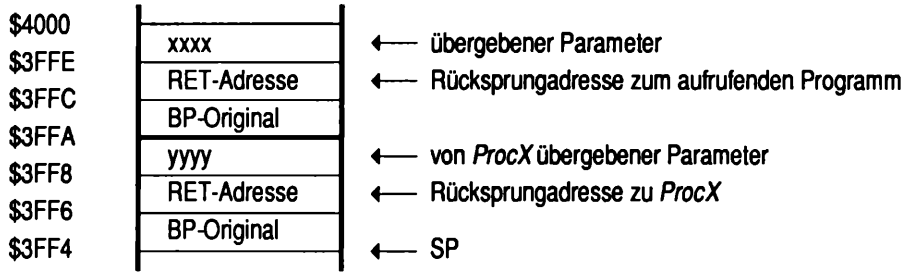
Noch einmal kurz zum Thema »Stack und Lokales«

Beim Aufruf einer Prozedur *ProcX*, die einen einzigen Parameter des Typs *Integer* und keine lokalen Variablen deklariert, sieht der Stack nach der Ausführung des Startcodes innerhalb von *ProcX* so aus:



*) Und kommt dabei ohne Rekursion aus. Mir ging es in diesem Kapitel um das Auffüllen der Lücke zwischen Einführungsbüchern und den Anforderungen, die *Algorithmen und Datenstrukturen* oder andere Bücher akademischen Niveaus an den Leser stellen. Bei allem Selbstbewußtsein als Autor: Prof. Wirth in seinen Erläuterungen übertreffen zu wollen, ist mir etwas zu hoch gegriffen.

Wenn die Routine nun ihrerseits eine Prozedur *ProcY* aufruft, die ebenfalls einen einzigen Parameter und keine lokalen Variablen deklariert, dann stapeln sich die Daten untereinander auf dem Stack:



Genau dasselbe ist der Fall, wenn *ProcX* anstelle von *ProcY* »sich selbst« aufruft. Ein Programmbeispiel einfachster Art dazu:

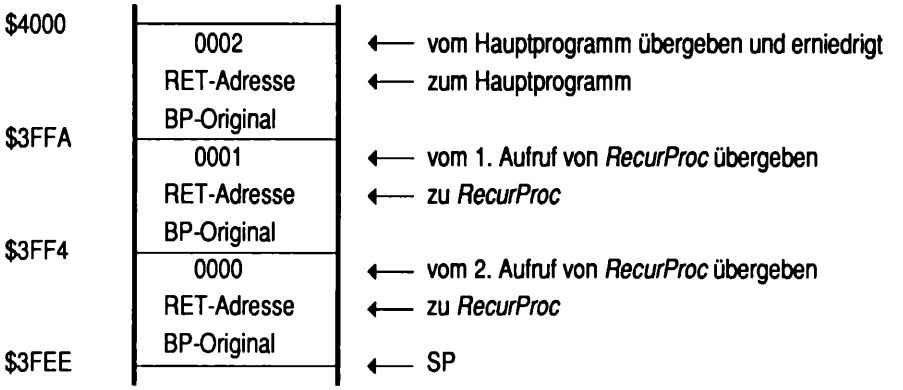
```
program RecurDemo1;

procedure RecurProc(x: Integer);      { Ausgaben des Programms:   }
begin                                 { Wert von x: 3             }
  Writeln('Wert von x: ', x);         { Wert von x: 2             }
  Dec(x);                             { Wert von x: 1             }
  if x > 0 then RecurProc(x);         { Wert von x: 1             }
  Writeln('Wert von x: ', x+1);       { Wert von x: 2             }
end;                                  { Wert von x: 3             }

begin
  RecurProc(3);
end.
```

Hier wird der übergebene Wert jeweils um eins erniedrigt - falls er noch größer als Null ist, führt *RecurProc* einen weiteren Aufruf von *RecurProc*[*] aus. Der Gesamtablauf läßt sich grafisch recht anschaulich darstellen:

```
'x = 3'
Aufruf   --> 'x = 2'
             Aufruf   --> 'x = 1'
                          <kein weiterer Aufruf>
                          'x = 1'
             'x = 2'  <-- RET
'x = 3'  <--  RET
```

[*] Der Terminus »ruft sich selbst auf« wurde hier absichtlich vermieden: *RecurProc* arbeitet bei jedem Aufruf mit anderen Parametern und lokalen Variablen - daß sich die Anweisungen »zufällig« auf derselben Adresse im Codesegment befinden, tut von der Logik her nichts zur Sache.

Der Stack sieht am »tiefsten Punkt« (d.h. nach der Erniedrigung von *x* auf 0) so aus:



Weitere Aufrufe von *RecurProc* unterscheiden sich von Aufrufen anderer Routinen nur dadurch, daß die RET-Adresse immer wieder auf denselben Punkt zeigt. Umgangssprachlich formuliert: Jeder weitere Aufruf fügt einen neuen Parameterblock zum Stack hinzu, Rücksprünge klettern den Stack Block für Block wieder hinauf. Da der Parameter *x* wie eine lokale Variable über BP adressiert wird und dieses Register für jede »Version« von *RecurProc* mit einem eigenen Wert arbeitet, greift der Prozessorbefehl zum Lesen von *x* (`MOV AX,[BP+04]`) auch jedesmal auf die »versionseigene« Adresse von *x* zu.

Regeln, Forderungen und Vorsichtsmaßnahmen

Eine rekursive Routine braucht (wie jede Art von Wiederholungsanweisung) ein *Abbruchkriterium*, das im Fall von *RecurProc* aus der Erniedrigung von *x* und einem anschließenden Test besteht. Wenn kein Abbruchkriterium vorhanden ist, wird der Stack solange mit immer neuen »Blöcken« beladen, bis er überläuft.

Achtung! Auch wenn es schon fast an eine Beleidigung grenzt, diesen Punkt explizit zu erwähnen: Speziell während der Testphase eines rekursiven Programms sollte die Stackprüfung immer aktiviert sein (`{$S+}`).

Wert-Parameter jeder Art, Funktionsergebnisse und lokal deklarierte *Variablen* belegen bei jedem weiteren Aufruf einen »versionseigenen« Bereich auf dem Stack - im Gegensatz zu allen Werten, die im Datensegment des Programms gespeichert sind (globale Variablen und typisierte Konstanten, letztere unabhängig vom Geltungsbereich ihrer Deklaration).

Bei Variablenparametern wird ein Zeiger übergeben, der für rekursive Aufrufe ebenfalls »versionseigen« auf dem Stack gespeichert ist. Die Variable, auf die dieser Zeiger zeigt, wird *nicht* lokal gespeichert, solange man nicht explizit (d.h. durch Zuweisung an eine echte lokale Variable) dafür sorgt.

Jeder Aufruf einer Stringfunktion reserviert $100 Bytes lokal als »temporären Speicherplatz« für das Ergebnis, jede Übergabe eines strukturierten Wertparameters reserviert Platz für eine »Lokalkopie«, die durch den Startcode der Routine

ausgeführt wird (siehe Kapitel 1). Wenn Sie mit großen Rekursionstiefen arbeiten, sollten Stringfunktionen deshalb soweit wie irgend möglich aus der rekursiven Routine ausgelagert und var-Parameter verwendet werden. Einige Beispiele dazu:

```
procedure RecursLang (s: string; x: Integer);
    { <=== hier fügt der Compiler Startcode ein, der eine
           Lokalkopie von s erzeugt und dabei $100 Bytes auf
           dem Stack belegt }

begin
  Writeln(s + chr(x));   { <== hier setzt der Compiler einen
                             Aufruf von ConCat ein, für dessen
                             "temporäres" Ergebnis $100 Bytes
                             auf dem Stack belegt werden - und
                             zwar permanent }
  if x-1 > 0 then RecursLang(s,x-1);
  ....
end;

{ ------------------------ }

procedure SxOut(var s: string, x: Integer);
{ "Ausgelagerte" Stringfunktion }
begin
  Writeln(s + chr(x));
end;

procedure RecursKurz(var s: string, x: Integer);
begin
   SxOut(s,x);    { Möglichkeit 1: keine weitere Belegung des Stacks }

   Writeln(s , chr(x));   { <== keine Stringfunktion, sondern zwei
                              voneinander getrennte Parameter! }
   if (x-1) > 0 then RecursKurz(s,x-1);
   ....
end;

{ ----------------------- }
procedure RecursMittel(s: string, x: Integer);
        { <== hier werden $100 Bytes für s belegt }
begin
   s[0] := chr(ord(s[0])+1);  { direkte Erhöhung der Länge }
   s[ord(s[0])] := chr(x);    { direktes Einfügen von chr(x) }
   Writeln(s);

   if x-1 > 0 then RecursMittel(s,x-1);
  ...
end;
```

Absuche von Directorystrukturen

Die »Kernroutine« von FILEFIND*) und einer ganzen Reihe anderer Programme der *Norton Utilities* ist wirklich das beste Demonstrationsbeispiel für Rekursion:

- die Daten (sprich: Dateieinträge) sind bereits vorhanden, namentlich bekannt und in ihrer Anzahl begrenzt;
- als »Debugger« steht der DOS-Befehl `dir` zur Verfügung;
- das Problem läßt sich iterativ (d.h. über Schleifen) nicht oder nur mit unverhältnismäßig hohem Aufwand lösen;
- mit dem Programm kann man hinterher etwas Sinnvolles anfangen.

A propos Problematik: Vor lauter Freude über die gefundene Demonstrationsmöglichkeit wäre ihre Erläuterung beinahe vergessen worden. Also: Gegeben sei eine Struktur von Dateieinträgen wie

```
DIR_0\
 Datei1
 Datei2
 DIR_A ──────→ DateiA_1
 Datei3        DateiA_2
 Datei4
(DIR_B ──────→ ...)
```

Der »instinktive« Lösungsansatz für eine Routine, die mit *DIR_0\Datei1* beginnt und sämtliche Einträge dieser Struktur überprüfen will, ist recht einfach: Der Eintrag *DIR_A* wird beim Lesen zwischengespeichert und zum Schluß bearbeitet (also nach *Datei3* und *Datei4*). Und falls sich in *DIR_0* mehr als ein weiteres Directory befindet, kann man schließlich auch ein Array zur Zwischenspeicherung »unbearbeiteter Directories« deklarieren.

Problematisch wird die Sache, wenn eines der in *DIR_0* verzeichneten Directories seinerseits ein Directory enthält:

```
DIR_0\
 Datei1
 Datei2
 DIR_A ──────→ DateiA_1
 Datei3        DIR_AA ──────→ DateiAA_1
 Datei4
 DIR_B ──────→ DateiB_1
```

*) Falls es tatsächlich Leser geben sollte, die dieses Programm noch nicht auf die eine oder andere Weise erworben haben: FILEFIND hat dieselbe Syntax wie der DOS-Befehl `dir`, sucht aber sämtliche Verzeichnisse eines Laufwerks ab.

Wenn das Programm hier nach der Bearbeitung von DIR_0 mit DIR_A beginnt, muß es sich einerseits merken, daß in DIR_0 noch ein weiteres Directory existiert (DIR_B), andererseits müssen wir DIR_AA innerhalb von DIR_A zwischenspeichern - es wird also ein zweidimensionales Array für unbearbeitete Directories gebraucht*).

Kurz und gut: Jede weitere Directory-Ebene macht auch eine weitere Arraydimension erforderlich - bereits die Auswertung einer Struktur wie

```
DIR_0\
 Datei1
 Datei2
 DIR_A  ---->  DateiA_1
 Datei3        DIR_AA  ---->  DateiAA_1
 Datei4                       DIR_A_A  ---->  Datei_AA1
 DIR_B  ---->  DateiB_1                       DIR_AAA  ---->  Datei_AA2
```

über die Zwischenspeicherung noch zu bearbeitender Directories ist eine mehr als unwirtschaftliche und ausgesprochen komplizierte Angelegenheit.

Der »Verwaltungsaufwand« der rekursiven Lösung besteht aus einer **if..then**-Anweisung:

```
procedure ScanDir (Path: string);
   ...
   while <Directory-Ende noch nicht erreicht> do
   begin
     <Eintrag> = NextEntry(Path);  { ersten/nächsten Eintrag lesen }
     if <Eintrag> = Directory
       then ScanDir (Path + <Eintrag>)
       else PrintEntry(<Eintrag>);
   end;
end;
```

Verfolgen wir einmal den Ablauf dieser Routine für die zuerst gezeigte Directorystruktur:

```
DIR_0\
 Datei1
 Datei2
 DIR_A  ---->  DateiA_1
 Datei3        DateiA_2
 Datei4
```

*) ... falls man an dieser Stelle nicht mit einem Pseudostack-Array anfängt - aber das wäre schließlich eine »zu Fuß« programmierte Art von Rekursion.

1. *ScanDir* wird (von wo aus auch immer) mit dem Parameter *DIR_0* aufgerufen. Innerhalb der ersten beiden Schleifendurchläufe liest die Routine die Dateinamen *Datei1* und *Datei2*, danach stößt sie auf das Directory *DIR_A*.

2. *ScanDir* ruft *ScanDir* mit dem Parameter *DIR_0\DIR_A* auf. Der ursprüngliche Wert für *Path* (*DIR_0*) bleibt dabei auf dem Stack. (Noch einmal: daß es sich hier »zufällig« um dieselbe Routine handelt, spielt überhaupt keine Rolle - der ursprüngliche Wert von *Path* würde auch beim Aufruf einer x-beliebigen anderen Routine erhalten bleiben).

   ScanDir liest die beiden Einträge *DateiA_1* und *DateiA_2*. Das Directory *DIR_A* ist damit zu Ende gelesen, *ScanDir* kehrt also zu dem Punkt zurück, von dem aus diese Routine aufgerufen wurde. Daß dieser Rücksprung wieder direkt in den Code von *ScanDir* hineinführt, spielt ebenfalls keine Rolle - wichtig ist nur, daß wir nun wieder mit *Path* = *DIR_0* weiterarbeiten.

3. Die Einträge *Datei3* und *Datei4* werden gelesen. Danach endet die Schleife (und damit auch die Ausführung der Routine).

Der Witz an diesem Verfahren liegt eben darin, daß es für beliebige Verschachtelungstiefen funktioniert (solange genügend Platz auf dem Stack ist). Ersparen wir uns den »Beweis« in Form einer schrittweisen Verfolgung der zuvor gezeigten Directorystruktur mit vier Ebenen und kommen wir zur Realisierung der Idee:

```
program DirScan1;   { 23/12/87 as }
{$S+}
{ Einfachste Demonstration einer rekursiven Absuche der
  Directory-Struktur des momentan gesetzten Laufwerks }

uses Dos;

procedure ScanDir(Path: string);
var
  SR: SearchRec;
begin
   FindFirst(Path+'\*.*',AnyFile,SR);   { ersten Eintrag suchen }
   while DosError = 0 do
   begin
     if (SR.Name[1] <> '.') then  { Einträge '.' und '..' ignorieren }
       if (SR.Attr and Directory) <> 0 then
         begin                          { Directory: }
           Writeln(Path+'\'+SR.Name);   { Ausgabe des vollen Namens }
           ScanDir(Path+'\'+SR.Name);   { Bearbeitung der Einträge }
         end
         else Writeln('   ', SR.Name);  { Datei: Ausgabe des Namens }
     FindNext(SR);
   end;
end;
```

```
{ ---------------------------------- }
var  StartPath: string;

begin
  GetDir(0,StartPath);   { momentan gesetztes Directory }
  Writeln('Start: ', StartPath);
  if StartPath[Length(StartPath)] = '\'    { für 'X:\' }
    then Dec(Byte(StartPath[0]));          { -> letztes '\' raus }
  ScanDir(StartPath);
end.
```

Gegenüber dem theoretischen Ansatz sind hier nur zwei Einzelheiten dazugekommen:

1. Die ersten beiden Dateieinträge eines Directories stehen für das Directory selbst ('.') und für das Directory, in dem dieses Directory verzeichnet ist ('..'). Abgesehen davon, daß diese beiden Einträge nicht auf dem Bildschirm erscheinen sollen, *müssen* sie ausgeklammert werden, weil sie ebenfalls das Dateiattribut *Directory* haben: Wenn wir uns momentan in einem Directory namens *DirX* befinden, dann

   • »erkennt« *ScanDir* den Eintrag '.' als *Directory* und ruft *ScanDir* mit dem Namen *DirX*. erneut auf.

   • Unglücklicherweise ist das für DOS ein legaler Dateiname - nämlich dasselbe wie *DirX*. Der Aufruf von *FindFirst* liest also wiederum den ersten Eintrag von *DirX*, (= '.') und »erkennt« ein Directory. Weiter geht es mit dem ersten Schritt - solange, bis der Stack voll ist und das Programm (hoffentlich) von der Prüfroutine der Laufzeitbibliothek abgebrochen wird.

2. Die Prozedur *GetDir* liefert den Suchweg zum momentan gesetzten Directory normalerweise ohne einen folgenden Schrägstrich (z.B. D:\TP40\PROGRAMS). Wenn das Stammverzeichnis als momentanes Directory gesetzt ist, liefert *GetDir* auch den Namen dieses Verzeichnisses zurück - und der ist nun einmal definitionsgemäß '\'. (Ohne die Korrektur würde *ScanDir* daraus Dateinamen wie *D:*.** konstruieren).

Für die Directory-Struktur

```
DIR_0\
 Datei1
 Datei2
 DIR_A  ──────→  DateiA_1
 Datei3          DateiA_2
 Datei4
```

eines Laufwerks *X* erhalten Sie von *DirScan1* die Ausgaben:

```
X:\DIR_0
  Datei1
  Datei2
X:\DIR_A
  DateiA_1
  DateiA_2
  Datei3      <-- diese beiden Dateien gehören aber
  Datei4      <-- eigentlich zu DIR_0
```

Soviel zum Prinzip. *ScanDir1* und die Ausführung des Suchalgorithmus sind in mindestens zwei Punkten verbesserungswürdig:

- Zwischen Dateien und Directorynamen wird nicht unterschieden. Eine Suche nach Dateien mit dem Suffix .PAS würde folglich auch nur Directories mit diesem Suffix berücksichtigen.
- Die Ausgaben von *ScanDir1* sind nicht nur unordentlich, sondern teilweise sachlich falsch, weil Subdirectories sofort bearbeitet werden. Bei einer reinen Suche würde die Reihenfolge keine Rolle spielen - da wir gefundene Dateieinträge aber über die Ausgabe dem jeweiligen Directory zuordnen, müssen Dateien und Directories *en bloc* erscheinen.

Wie das nächste Programm zeigt, lassen sich beide Fliegen erfreulicherweise mit einer Klappe schlagen.

Rekursion und Schleifen kombiniert

Das Programm *FFClone* arbeitet nach demselben Prinzip wie *ScanDir1*, verwendet aber *zwei Schleifen* innerhalb der rekursiven Prozedur:

```
procedure ScanDirectory(Path,FName: string; Attribute: Word);
var
  SR: SearchRec;
begin
  { Teil 1: Suche nach Dateieinträgen }
  if Attribute and Directory <> 0     { falls Attribut 'AnyFile' }
     then Dec(Attribute,Directory);
  FindFirst(Path+'\'+FName,Attribute,SR);
  while DosError = 0 do
  begin
    ProceedEntry(SR);    { Ausgabe (oder was auch immer) des Eintrags }
    FindNext(SR);
  end;
```

Innerhalb der ersten Schleife wird das jeweilige Directory nur nach normalen Dateieinträgen abgesucht - Subdirectories werden auch dann ignoriert, wenn ihr Name mit der als *FName* übergebenen Suchmaske übereinstimmt.

Der Aufruf der Prozedur *ProceedEntry* zusammen mit den Informationen über den gefundenen Eintrag ist eine Verallgemeinerung - da erst in *ProceedEntry* entschieden wird, was mit dem Eintrag passiert, läßt sich das »Kernstück« *ScanDirectory* ohne weitere Modifikationen für die verschiedensten Zwecke einsetzen. (Das Program DATEFIND im nächsten Abschnitt macht von dieser Möglichkeit Gebrauch).

```
    { Teil 2: Suche nach Subdirectories }
    FindFirst(Path+'\*.*',Directory,SR);
    while DosError = 0 do
    begin
      if (SR.Attr and Directory <> 0) and (SR.Name[1] <> '.') then
      begin    { nicht der eigene oder "Parent"-Eintrag }
        DirDisplay(Path,SR);  { Namensausgabe }
        ScanDirectory(Path+'\'+SR.Name,FName,Attribute); { Bearbeitung }
      end;
      FindNext(SR);
    end;
  end;
```

Die zweite Schleife von *ScanDirectory* sucht dasselbe Directory noch einmal von vorne ab - diesmal wird ausschließlich nach Subdirectories gefahndet.

Aufgrund dieser Trennung ist das Programm von der Suchmaske für Dateinamen und eventuell angegebenen Attributen unabhängig: für die Suche nach Subdirectories wird immer der Dateiname '*.*' benutzt.

Außerdem können wir die Ausgaberoutinen für Dateinamen und Subdirectories recht einfach voneinander trennen: Anstelle von *ProceedEntry* wird in der zweiten Schleife die Prozedur *DirDisplay* für die Ausgabe gefundener Einträge aufgerufen.

Das vollständige Programm enthält eine ganze Reihe zusätzlicher Verzierungen:

- Der Name des Directories, das momentan abgesucht wird, erscheint auf dem Bildschirm*) - der Benutzer kann die Arbeit des Programms also auch dann verfolgen, wenn die Suche mehr oder weniger erfolglos bleibt.
- Gefundene Dateieinträge werden über die Prozedur *DisplayEntry* zusammen mit sämtlichen Informationen (Daten, Größe, Attribute usw.) ausgegeben. *DisplayEntry* ist in einer eigenen Include-Datei gehalten, weil wir diese Routine noch in einem weiteren Programm verwenden.
- *FFClone* zählt über eine globale Variable mit, wieviele Dateieinträge die Suche zutage gefördert hat, und gibt das Ergebnis am Ende der Suche aus.

*) Was eine ausgesprochen beruhigende Sache ist - bei einer erfolglosen Suche auf einer wirklich vollen Festplatte steht der Benutzer sonst bis zu 20 Sekunden im Dunklen. Peter Norton hat wahrscheinlich zugunsten der »Allgemeinkompatibilität« auf dieses Feature verzichtet.

- Das Programm verhält sich in Bezug auf den angegebenen Dateinamen wie der DOS-Befehl `dir` (keine Angabe -> '*.*'; Name ohne Suffix -> Name.* usw).
- Das Rollen des Bildschirms bei der Ausgabe von Dateinamen läßt sich durch einen beliebigen Tastendruck anhalten und mit einem zweiten Tastendruck wieder fortsetzen. (Der Abbruch des Programms ist dagegen nur über CONTROL-BREAK bzw. CONTROL-C möglich).

Weil es niemanden interessiert, in welchen Directories das Programm *nichts* gefunden hat, wechselt die Anzeige des »momentanen« Directories innerhalb einer Bildschirmzeile - *FFClone* kommt deshalb ohne das Unit *Crt* und einige zusätzliche boolsche Variablen nicht aus. Was leider auch heißt, daß das Programm nur im Textmodus sichtbare Ergebnisse zeitigt.

```
program FFClone;   { 12/87, 12/88 as }
{ Demonstration einer rekursiven Absuche der Directory-
  Struktur des momentan gesetzten Laufwerks -
  ein leicht verschönerter Clone von Peter Nortons FILEFIND. }

uses Crt,Dos;

var
  x: Integer;          { für das Anhängen/Einfügen von '*' in den Dateinamen }
  RootName: string[3];             { für den Namen des Stammverzeichnisses }
  SearchName: string;              { Suchmaske }
  FoundEntries: Boolean;           { "Einträge in einem Directory gefunden" }
  FileCount: Word;     { Zähler für gefundene Dateieinträge }

{$I DSPENTRY.INC}    { DisplayEntry & CheckStop - Ausgabe gefundener Dateien }
```

DSPENTRY.INC bzw. die Prozeduren *DisplayEntry* und *CheckStop* sind im nächsten Abschnitt gelistet.

```
procedure ProceedEntry(var SR: SearchRec);
{ Bearbeitung eines gefundenen Dateieintrags }
begin
  Writeln;             { vorherigen Namen abschließen }
  DisplayEntry(2,SR);  { 2 Leerzeichen voranstellen, Ausgabe }
  FoundEntries := True;
  Inc(FileCount);      { Zähler für gefundene Dateieinträge }
end;
```

```
procedure DirDisplay(var Path:string; var SR: SearchRec);
{ Ausgabe eines neuen Directorynamens }
begin
  if FoundEntries then
  begin
    Writeln;     { vorherigen Dateieintrag abschließen }
    Writeln;     { eine Leerzeile }
  end
  else GotoXY(1,WhereY);  { ansonsten zurück auf den Zeilenanfang }

  CheckStop;     { in DSPENTRY.INC - Prüfung auf ^C, Anhalten }
  Write(Path+'\'+SR.Name); ClrEol;
  FoundEntries := False;  { = noch keine Einträge in diesem neuen Directory }
end;
```

Bei der Ausgabe eines Directorynamens müssen wir unterscheiden, ob die Absuche des vorigen Directories Dateieinträge zutage gefördert hat oder nicht. Im ersten Fall wird diese Ausgabe mit einem Zeilenvorschub abgeschlossen, auf den eine weitere Leerzeile folgt - im zweiten Fall wird der »alte« Directoryname durch den neuen ersetzt.

```
procedure ScanDirectory(Path,FName: string; Attribute: Word);
var SR: SearchRec;
begin
  { Teil 1: Suche nach Dateieinträgen }
  if Attribute and Directory <> 0 then Dec(Attribute,Directory);
  FindFirst(Path+'\'+FName,Attribute,SR);
  while DosError = 0 do
  begin
    ProceedEntry(SR);   { Ausgabe (oder was auch immer) des Eintrags }
    FindNext(SR);
  end;

  { Teil 2: Suche nach Subdirectories }
  FindFirst(Path+'\*.*',Directory,SR);
  while DosError = 0 do
  begin
    if (SR.Attr and Directory <> 0) and (SR.Name[1] <> '.') then
    begin    { nicht der eigene oder "Parent"-Eintrag }
      DirDisplay(Path,SR);       { Ausgabe des Namens }
      ScanDirectory(Path+'\'+SR.Name,FName,Attribute); { Bearbeitung }
    end;
    FindNext(SR);
  end;
end;
```

Die Routine *ScanDirectory* ist mit dem Listing am Anfang dieses Abschnitts identisch.

```
  { ================================ }
begin
  Writeln('FFClone - (c) 12/87 as');   { <-- sonst funktioniert    }
  Writeln;                             {     das Programm nicht... }
  if ParamCount = 0 then SearchName := '*.*'
   else
   begin
     SearchName := ParamStr(1);
     x:= Pos('.', SearchName);
     if x = 0 then
       SearchName := SearchName + '.*'   { kein Suffix: .* anhängen }
      else if x = 1 then
        SearchName := '*' + SearchName;  { nur Suffix: '*' voranstellen }
   end; { if ParamCount = 0 }

  FoundEntries := False; { = bis jetzt keine Einträge im }
  FileCount := 0;            { Stammverzeichnis gefunden }

  GetDir(0,RootName); Write(RootName);    { "erstes Directory" }
  ScanDirectory('',SearchName,AnyFile);
  if not FoundEntries then
  begin
    GotoXY(1,WhereY); ClrEol;  { letzten Directory-Namen löschen }
  end
   else Writeln;  { letzten Eintrag abschließen }
  Writeln;
  Writeln(FileCount, ' Dateien.');
end.
```

Weil das Stammverzeichnis nirgendwo als Eintrag verzeichnet ist (bzw. *ScanDirectory* nur auf »echte« Dateinamen reagiert), muß dieser Name vom Hauptprogramm ausgegeben werden. Der Trick, *RootName* als `string[3]` zu deklarieren und damit einen Aufruf von *Copy* einzusparen, dürfte zwar nicht unbedingt das Schönste an Programmierstil sein, spart aber außerdem noch Speicherplatz.

Die Ausgabe der Anzahl gefundener Dateieinträge am Ende des Programms folgt demselben Prinzip wie die Routine *DirDisplay*: Wenn die Suche innerhalb des zuletzt bearbeiteten Directories erfolglos war (*FoundEntries* = *False*), wird der Directoryname gelöscht; ansonsten schließt der Aufruf von *Writeln* die letzte Dateinamen-Ausgabe ab.

Die gezeigte Version von *FFClone* bietet vier Ansatzpunkte für Erweiterungen:

- Das Programm reagiert nicht sonderlich sinnvoll auf Kommandozeilen-Parameter, die einen vollständigen oder unvollständigen Suchweg enthalten - es findet in diesen Fällen einfach nichts.
- Die Suche beginnt grundsätzlich mit dem Stammverzeichnis.

- Der Parameter *Attribute* wird in *ScanDirectory* zwar mitgeschleppt, das Hauptprogramm setzt ihn aber grundsätzlich auf *AnyFile*. Die Prüfung für »archivierte« Dateien über den Vergleich von *SR.Attr* mit 0 funktioniert zwar für diesen Fall, ist aber alles andere als allgemein verwendbar. (Überhaupt könnte man aus *Attribute* eigentlich eine globale Variable machen).
- Das Programm bearbeitet immer das momentan gesetzte Laufwerk.

Die Ausgabe eines Dateieintrags: DSPENTRY.INC

Die Routine *DisplayEntry* entspricht im großen und ganzen der in Kapitel 6 des Benutzerhandbuchs besprochenen Ausgaberoutine für Dateieinträge*). Sie gibt zusätzlich die Attribut-Bits des Eintrags aus und prüft nach jeder Ausgabe, ob der Benutzer eine Taste gedrückt hat.

```
{ DSPENTRY.INC -- Prozedur DisplayEntry für FFClone und DateFind }
const
  Attributes: array[0..5] of Char = ('R','H','S','V','D','A');

procedure CheckStop;   { Prüfung auf ^C, wird auch von DirDisplay aufgerufen }
begin
   if KeyPressed then          { Prüfung auf einen Tastendruck }
   begin                       { des Benutzers }
     repeat
       if ReadKey = #3 then    { Control-C? }
          begin
            GotoXY(1,WhereY);
            Writeln(#13#10'^C');     { Anzeige des Abbruchs }
            Halt;
          end;
     until (not KeyPressed);   { sonst alle Tastendrücke abholen und }
     if ReadKey = #0 then ;    { dann auf einen weiteren Tastendruck warten }
   end;
end;

procedure DisplayEntry(Blanks: Integer; SR: SearchRec);
{ Gibt einen Dateieintrag mit Größe, Datum und Uhrzeit aus }
var x: Integer;
    DTRec: DateTime;

 function TwoDig(Arg:Integer):string;
 { Liefert eine Zahl als String mit zwei Zeichen (und eventuell einer
   führenden Null) zurück. }
 begin
   TwoDig := Chr(Ord('0') + Arg div 10) + Chr(Ord('0') + Arg mod 10);
 end;
```

*) Wenn man von einigen Straffungen absieht, die ich den Lesern des Benutzerhandbuchs nicht zumuten wollte (wie z.B. die gepackte Ausgabe von Datum und Uhrzeit).

```
begin  { DisplayEntry }
  with SR do
  begin
    if Blanks >= 1 then Write(' ':Blanks);  { Leerzeichen davor }
    x:= Pos('.',Name);  { Suffix? }
    if x = 0 then x := Length(Name)+1;
    Write(Copy(Name,1,x-1),' ':10-x);      { Dateiname }
    Write(Copy(Name+'   ',x+1,3):5);  { Suffix }
    Write(Size:8);     { Dateigröße in Bytes }
  end;

  UnpackTime(SR.Time,DTRec);   { Datum/Uhrzeit }

  with DTRec do  { Hier stehen bereits Datum & Uhrzeit der Datei }
  begin
    Write(TwoDig(Day):5,'-',TwoDig(Month),'-',TwoDig(Year mod 100));
    Write(TwoDig(Hour):5,':',TwoDig(Min));
  end;  { with SR }

  Write(' ':5);
  with SR do
    for x:= 5 downto 0 do          { Ausgabe der Dateiattribute }
      if Attr and (1 shl x) <> 0   { $20 $10 $08 $04 $02 $01 }
        then Write(Attributes[x]:2)
        else Write('  ');
  CheckStop;   { Prüfung auf Tastendruck, ^C }
end;  { DisplayEntry }
```

DATEFIND

Ist es Ihnen auch schon einmal passiert, daß Sie die Vorgabe eines Programms für einen Dateinamen versehentlich mit [a][↵] (oder, je nach Tastaturlayout) mit einer anderen unerwünschten Tastenkombination »bestätigt« haben? Nicht nur alle Turbo-Pakete, sondern auch WORD und eine ganze Reihe anderer Programme löschen daraufhin ihre Vorgabe, akzeptieren das versehentlich eingegebene Zeichen als neuen Dateinamen und schreiben die Datei in das momentan gesetzte Verzeichnis. Wer das Programm dann beendet, ohne noch einmal auf den Bildschirm zu gucken, der hat entweder Glück (die ursprünglichen Daten stammen ebenfalls aus diesem Verzeichnis, das nicht allzuviele Einträge enthält) - oder er tut gut daran, sich zu erinnern, welches Verzeichnis zum Zeitpunkt der Speicherung als Standard gesetzt war.

Dieselbe Problematik ergibt sich bei der Entwicklung größerer Programme, bei der anscheinend zwangsläufig immer neuere Versionen mit immer absonderlicheren Namen entstehen: ich kenne niemanden, der bei einer derartigen Arbeit nicht mindestens ein- oder zweimal mit `dir` und dem Finger am Bildschirm nach der Version sucht, die er als vorvorletztes... wie hieß sie doch noch gleich?

Kurz: Bei allen Computern der PC-Serie bekommt zwar jede neu erzeugte oder veränderte Datei die Systemzeit zugewiesen - einen Befehl aber, mit dem man Dateien über dieses Kriterium suchen könnte, haben die Entwickler augenscheinlich vergessen. Nicht einmal Peter Norton hat daran gedacht*).

Wie nach dieser Einleitung wohl zu erwarten, ist DATEFIND die Lösung des Problems - ein Programm, das die Festplatte nach Dateien absucht, die im Laufe der letzten *x* Minuten verändert oder neu erzeugt wurden.

Die Routinen zur Absuche der Directory-Struktur (*ScanDirectory*), der Ausgabe von Directorynamen (*DirDisplay*) und der Ausgabe von Dateieinträgen, die den Suchkriterien entsprechen (*DisplayEntry* in der Include-Datei DSPENTRY.INC) sind absolut dieselben wie in *FFClone*.

Geändert haben sich: das Hauptprogramm, weil *DateFind* mehrere Parameter unterstützt und die Entscheidung in *ProceedEntry*, wann ein gefundener Dateieintrag auch tatsächlich ausgegeben wird. Außerdem sind einige neue (und recht umfangreiche) Routinen zur Be- und Umrechnung von Datum und Uhrzeit hinzugekommen.

```
program DateFind;   { 23/12/87 as }
{ Eine Anwendungsmöglichkeit der rekursiven Absuche von
  Directory-Strukturen: Suche nach Dateien über Uhrzeit
  und Datum.

  Aufruf des Programms:  DateFind [Dateiname.ext] [/xx] [/A]

  Gibt alle Dateien aus, die innerhalb der letzten <xx> Minuten
  verändert oder neu erzeugt wurden.
  Wenn Dateiname.ext nicht angegeben ist, wird *.* eingesetzt;
  wenn /xx nicht angegeben ist, wird die Standardvorgabe 10
  eingesetzt. Wenn /A angegeben ist, wird das momentan gesetzte
  Laufwerk komplett abgesucht, ansonsten findet die Suche nur
  innerhalb des momentan gesetzten Directories und eventueller
  Subdirectories statt.
}

uses Crt,Dos;

const        { Zur Rückwärtsberechnung in das jeweilige Vorjahr }
  MonDays: array[1..12] of LongInt= (31,28,31,30,31,30,31,31,30,31,30,31);

var
  x: Word; Code: Integer;
  StartPath: string;
  ArgStr: string;                { Temp für Kommandozeilen-Parameter }
  SearchName : string;           { Dateiname (oder *.*) }
```

*) Ha!

```
  SNameSet: Boolean;              { "Dateiname gesetzt" }
  ParamsOk: Boolean;              { "Kommandozeilen-Parameter o.k." }
  FoundEntries: Boolean;          { "Einträge in einem Directory gefunden" }
  DTRec: DateTime;                { Datum/Uhrzeit für "jetzt" & Dateieintrag }
  DTNow: LongInt;                 { Datum/Uhrzeit aktuell in Minuten }
  YearNow: Integer;               { momentanes Jahr }
  TimeSub: Word;                  { Parameter <xx>  (s.o.) }
  FileCount: Word;                { Anzahl gefundener Dateien }

function ToMinutes(var DTimeRec: DateTime): LongInt;
{ Rechnet Monat, Tag, Stunde + Minute in eine Minutenzahl um. }
var Total: LongInt;
begin
   with DTRec do
   begin
     Total:= 0;
     for x:= 1 to Month-1 do
       Inc(Total,MonDays[x]);     { Anzahl der Tage seit Jahresanfang }
     Inc(Total,Day-1);            { Tage im momentanen Monat dazu }
     Total := Total * 24 + Hour; { = Anzahl der Stunden }
     Total := Total * 60 + Min;  { = Anzahl der Minuten }
   end; { with DTimeRec }
  ToMinutes:= Total;
end;

{$I DSPENTRY.INC }   { Prozeduren DisplayEntry = Ausgabe, CheckStop }
```

Den Aufruf von *UnpackTime* in *DisplayEntry* könnte man in diesem Fall hinauskommentieren, weil *DTRec* bereits durch den Vergleich in *ProceedEntry* gesetzt wird.

```
procedure ProceedEntry(var SR: SearchRec);
{ Ausgabe eines gefundenen Dateieintrags }
begin
  UnpackTime(SR.Time,DTRec);   { Datum -> DTRec }
  if DTRec.Year >= YearNow then
    if ToMinutes(DTRec) >= DTNow then
    begin
      Writeln;            { vorherigen Namen abschließen }
      DisplayEntry(2,SR);
      FoundEntries := True;
      Inc(FileCount);    { Anzahl gefundener Dateien }
    end;
end;
```

ProceedEntry vergleicht hier Datum und Uhrzeit eines gefundenen Dateieintrags mit dem Wert *DTNow*, der der momentanen Systemzeit minus xx Minuten entspricht. Die Einheit für diesen Vergleich ist »Minuten seit Beginn des Jahres«.

```
procedure DirDisplay(var Path:string; var SR: SearchRec);
{ Ausgabe eines neuen Directorynamens }
begin
  if FoundEntries then
  begin
    Writeln;     { vorherigen Dateieintrag abschließen }
    Writeln;     { eine Leerzeile }
  end
  else GotoXY(1,WhereY);  { ansonsten zurück auf den Zeilenanfang }

  CheckStop;     { Prüfung auf Tastendruck, ^C }
  Write(Path+'\'+SR.Name); ClrEol;
  FoundEntries := False;
end;

procedure ScanDirectory(Path,FName: string; Attribute: Word);
var
  SR: SearchRec;
begin
  { Teil 1: Suche nach Dateieinträgen }
  if Attribute and Directory <> 0 then Dec(Attribute,Directory);
  FindFirst(Path+'\'+FName,Attribute,SR);
  while DosError = 0 do
  begin
    if (SR.Attr and (Attribute) <> 0) or (SR.Attr = 0) { "Archive" }
       or (SR.Attr and Directory <> 0) then
      ProceedEntry(SR);    { Ausgabe (oder was auch immer) des Eintrags }
    FindNext(SR);
  end;

  { Teil 2: Suche nach Subdirectories }
  FindFirst(Path+'\*.*',Directory,SR);
  while DosError = 0 do
  begin
    if (SR.Attr and Directory <> 0) and (SR.Name[1] <> '.') then
    begin    { nicht der eigene oder "Parent"-Eintrag }
      DirDisplay(Path,SR);
      ScanDirectory(Path+'\'+SR.Name,FName,Attribute); { Bearbeitung }
    end;
    FindNext(SR);
  end;
end;
```

DirDisplay und *ScanDirectory* wurden gegenüber *FFClone* nicht verändert.

```
begin
  { Der folgende Aufruf von Writeln *muß* aus unerklärlichen Gründen
    mit aufgenommen werden - ansonsten formatiert das Programm die
    Festplatte oder stellt noch Schlimmeres mit Ihrem Computer an. }

  Writeln('DATEFIND - (c) as 12/87');

  with DTRec do        { Ermittlung des momentanen Datums und der }
  begin                { momentanen Uhrzeit }
    GetDate(Year,Month,Day,x); { x für DayofWeek <- Dummy }
    GetTime(Hour,Min,Sec,x);   { x für Sec100 <- Dummy }
   end;
   GetDir(0,StartPath);  { Annahme: Suche ab momentanem Directory }
   TimeSub := 10;
   SearchName := '*.*';        { Standardvorgabe }
   SNameSet := False;
```

Der erste Teil des Hauptprogramms: Ermittlung des Systemdatums und Setzen der Standardvorgaben für Suchweg, Suchmaske und die Anzahl der »vergangenen« Minuten.

```
   ParamsOk:= (ParamCount <= 3);       { Flag: "Aufruf korrekt" }
   for x:= 1 to ParamCount do
   begin
     ArgStr := ParamStr(x);
     if ArgStr[1] = '/' then
     begin
       if UpCase(ArgStr[2]) = 'A'
         then StartPath := Copy(StartPath,1,3)  { /A }
         else begin
                Val(Copy(ArgStr,2,255),TimeSub,Code);   { /xx }
                ParamsOk := (Code = 0);
                TimeSub := Abs(TimeSub);
            end;
     end
     else begin   { kein / als erstes Zeichen }
            if SNameSet then ParamsOk:= False; { 2. "Dateiname"! }
            SearchName := ArgStr;
            SNameSet := True;
          end;
   end;  { for x }
   if not ParamsOk or (ParamStr(1) = '?') then
   begin
     Writeln;
     Writeln('Aufruf: DATEFIND [Dateiname.ext] [/xx] [/A]');
     Writeln('xx = Anzahl Minuten (Standard 10), ',
             '/A = Suche ab Stammverzeichnis');
     Halt(1);
   end;
```

Die Überprüfung der Parameter und ihre Auswertung ist hier recht komplex, weil wir es mit wechselnden Parameterzahlen und -reihenfolgen zu tun haben. Die Prüfung über die boolsche Variable *SNameSet* verhindert, daß das Programm bei Aufrufen mit ungültigen Parametern wie `datefind *.pas x` den ungültigen Parameter x erneut als Suchmaske interpretiert.

Die (bei mir) übliche Methode, beim Aufruf des Programms mit einem Fragezeichen als Parameter eine Hilfestellung ausgeben zu lassen (`if ParamStr(1) = '?'`), ist hier eine etwas zweischneidige Angelegenheit: Wenn Sie diese Prüfung übernehmen, müssen Sie für die Suche nach Dateien mit nur einem Buchstaben als Namen mindestens einen weiteren Parameter *vor* dem Dateinamen angeben (z.B. DATEFIND /10 ?).

```
x:= Pos('.', SearchName);
  if x = 0 then
    SearchName := SearchName + '.*'    { kein Suffix: .* anhängen }
   else if x = 1 then
     SearchName := '*' + SearchName;  { nur Suffix: '*' voranstellen }
```

Die »übliche« Bearbeitung eines Dateinamens, der als Kommandozeilen-Parameter übergeben wurde.

```
{ Umrechnung von Datum/Uhrzeit in Minuten und Abziehen von TimeSub }
DTNow := ToMinutes(DTRec);
YearNow := DTRec.Year;

Dec(DTNow, TimeSub);

if DTNow < 0 then
begin     { Das hat gerade noch gefehlt... Jahresanfang }
  Dec(YearNow);
  for x:= 1 to 12 do
    Inc(DTNow,MonDays[x]*24*60); { zurück ans Ende des letzten Jahres }
end;
```

Umrechnung der Systemzeit in »Minuten ab Jahresanfang« und Subtraktion des Wertes xx. Wenn sich dabei ein Wert kleiner 0 ergibt, muß das Programm ins vorherige Jahr zurückrechnen - eine Operation, die sich in der gezeigten Version für Schaltjahre um einen Tag verzählt*).

```
with DTRec do
  Writeln('Suche nach Dateien mit Veränderungsdatum ab ',
          Day,'-',Month,'-',Year mod 100,
          ' ',Hour,':',Min,' - ',TimeSub,' Minuten');
```

*) Das nächste Schaltjahr kommt zwar bestimmt - aber wenn man sich diese Unmenge von Manipulationen für das Abziehen von ein paar Minuten ansieht, dann wird langsam klar, warum Betriebssysteme wie UNIX Datum und Uhrzeit gleich in Sekunden speichern und nur für Ausgaben umrechnen.

```
  { ---------------------------------------------------------- }

    FoundEntries := False; FileCount := 0;

    Writeln; Write(StartPath);
    if StartPath[Length(StartPath)] = '\' then   { endet mit '\' }
      Dec(Byte(StartPath[0]));
    ScanDirectory(StartPath,SearchName,AnyFile);
    if not FoundEntries then
    begin
      GotoXY(1,WhereY); ClrEol;  { letzten Directory-Namen löschen }
    end
    else Writeln;
    Writeln(FileCount, ' Dateien.');
  end.
```

Wie bei *FFClone* muß das Hauptprogramm die Ausgabe des ersten Directorynamens übernehmen und dafür sorgen, daß dieser Name nicht mit einem umgekehrten Schrägstrich endet.

Die Ausgabe der Anzahl gefundener Dateieinträge am Ende des Programms folgt demselben Prinzip wie die Routine *DirDisplay*: Wenn die Suche innerhalb des zuletzt bearbeiteten Directories erfolglos war (*FoundEntries* = *False*), wird der Directoryname gelöscht; ansonsten schließt der Aufruf von *Writeln* die letzte Dateinamen-Ausgabe ab.

Lineare dynamische Listen

Ein Zeiger-Array wie das in *StrSort2* (Seite 181) verwendete ist selbst dann nicht die ideale Lösung, wenn es um statische Datenmengen geht: Nach einer Deklaration wie

```
type
  LineStr: string[80];
var
  Lines: array[1..500] of ^LineStr;
```

belegt jeder Aufruf der Form `New(Lines[x])` dieselbe Menge an Speicherplatz (81 Bytes). Solange es auf einfache Programmierung und/oder Geschwindigkeit ankommt, ist dieses Verfahren recht brauchbar - aber: selbst BASIC-Interpreter gehen sparsamer mit dem Speicherplatz um.

Strings und die Prozedur *GetMem*

Auch wenn Ihnen der folgende Lösungsansatz etwas exotisch vorkommen mag - er stellt die einzige Möglichkeit dar*): Wir deklarieren ein Array mit Zeigern auf `string` und verwenden die Prozedur *GetMem* anstelle von *New*.

```
var
  Lines : array[1..500] of ^string;  { 500 Zeiger }
```

Die dynamische Speicherung eines Strings *s* über den Zeiger *x* des Arrays erfolgt mit

```
....
  s := 'Zu speichernder String';
  GetMem(Lines[x],Length(s)+1);  { + Platz für das Längenbyte }
  Lines[x]^ := s;
```

Strings enthalten bekanntlich ihre eigene Länge - über die Anweisung

```
  FreeMem(Lines[x], Length(Lines[x]^)+1);
```

wird man also eine solche Variable wieder los.

Vorsicht! Für jeden String ist hier exakt soviel Speicherplatz reserviert, wie tatsächlich benötigt wird. Längenveränderungen gespeicherter Strings sind nicht erlaubt:

- die Verlängerung eines Strings überschreibt Teile des Heaps, die unter Umständen für andere dynamische Variablen reserviert sind;
- die Verkürzung eines Strings erniedrigt (naturgemäß) das Längenbyte entsprechend. Ein Aufruf von *FreeMem* zu einem späteren Zeitpunkt würde deshalb weniger Speicherplatz freigeben, als mit *GetMem* für diese Variable belegt wurde.

Noch ein **Vorsicht!** In die Klasse »Längenveränderungen« fallen auch direkte Zuweisungen. Allgemein: jede Veränderung eines via *GetMem* gespeicherten Strings *muß* über eine statische Variable und in drei Schritten erfolgen

```
WorkStr := Lines[x]^;   { 1. Kopie und Freigabe }
FreeMem(Lines[x], Length(Lines[x]^) + 1);
   ....
         { 2. Veränderung von WorkStr, d.h. Bearbeitung des Strings }
   ....
GetMem(Lines[x],Length(WorkStr)+1);  { 3. Neubelegen und Speicherung }
Lines[x]^ := WorkStr;
```

*) a) die *mir* eingefallen ist; b) Records mit einem eigenen Feld für die tatsächlich belegte Bytezahl lösen nur das kleinste der drei im folgenden beschriebenen Probleme (nämlich das der Längenverkürzung) und sind wesentlich umständlicher zu bearbeiten.

Und noch eins: **Vorsicht!** Wenn die Zeiger von *Lines* für einen kürzeren Stringtyp deklariert sind, begrenzt der Compiler (wie bei normalen Strings) die Länge von Zuweisungen automatisch*). Ein Beispiel:

```
type Str1: string[1];
var p1: ^Str1;
begin
  GetMem(p1, 200);
  p1^:= 'Funktioniert!';
  Writeln(p1^);      { ergibt F - und nicht mehr }
end.
```

Für die Veränderung von *Zeigern auf Strings* gelten diese Grundsätze nicht, weil der Heap in diesem Fall unverändert bleibt. Der Austausch zweier Stringvariablen wie im Programm *StrSort2* geschieht also nach wie vor mit

```
TempPtr := Lines[x];     { Vertauschen der Strings [x] und [x+1] }
Lines[x] := Lines[x+1];
Lines[x+1] := TempPtr;
```

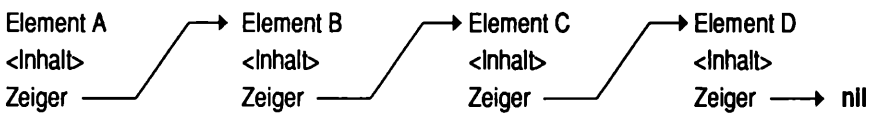
Was das eigentlich mit dynamischen Listen zu tun hat? Nun - eine verkettete Liste, deren Elemente als

```
type
  ListPtr = ^ListElement;
  ListElement = record
                  Line: ^string;  { Inhalt }
                  Next: ListPtr;  { Zeiger zum nächsten Element }
                  { ...   }
                end;
```

definiert sind, hat mit demselben Problem zu kämpfen. Damit die folgenden Beispiele nicht komplizierter werden, als sie sowieso schon sind, werden wir allerdings mit fixen Stringlängen arbeiten - die Möglichkeit der Belegung über *GetMem* ist an den entsprechenden Punkten jeweils nur angedeutet.

Einfache Listen...

heißen nur deshalb »einfach«, weil die einzelnen Elemente der Liste über jeweils einen Zeiger miteinander verbunden sind. Nachdem der Prototyp von Listen dieser Art bereits in Kapitel 6 des Benutzerhandbuchs recht ausführlich besprochen ist, beschränken wir uns auf eine kurze Wiederholung der Struktur:



*) vgl. »Variablengrößen und -strukturen« im ersten Teil dieses Kapitels.

Alle Elemente dieser Liste haben den gleichen Typ - einen Record, der mindestens zwei Felder enthält:

```
type
  ListPtr = ^ListElement;
  ListElement = record
                  Inhalt:   < Datentyp >
                   ...              { weitere Felder mit Daten }
                  Next: ListPtr;    { Zeiger zum nächsten Element }
                end;
```

Das Programm definiert einen statischen Zeiger (*LRoot*) auf das erste Element der Liste (A), der die einzige Zugriffsmöglichkeit darstellt. Der Zugriff auf das Element C der Liste erfolgt mit:

```
p := LRoot;        { p auf die Adresse von A }
p := p^.Next;      { liest den Zeiger "Next" des Elements A, d.h.
                     die Adresse von B }
p := p^.Next;      { liest den Zeiger "Next" des Elements B, d.h.
                     die Adresse von C }
```

Diese Anweisungsfolge läßt sich auch allgemein darstellen: Der Zugriff auf das Element N einer verketteten Liste geschieht mit:

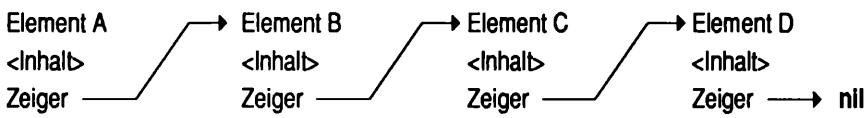
```
p := LRoot;            { p auf das erste Element }
for x:= 1 to N-1 do
   p := p^.Next;       { bewegt p N-1 Elemente weiter }
```

Einfügen und Löschen von Elementen

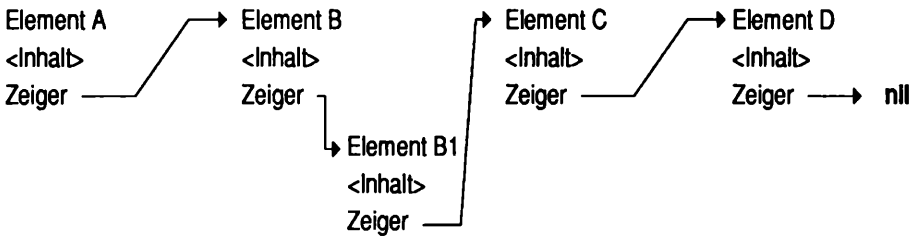
Eine verkettete Liste hat gegenüber einem Zeiger-Array einen Nachteil (nämlich den, daß man sich für den Zugriff auf das Element N erst N-1 Elemente »entlanghangeln« muß) und zwei gewaltige Vorteile:

- die Anzahl der Listenelemente ist theoretisch unbegrenzt (d.h. nur von der Größe des Hauptspeichers abhängig);
- da die Reihenfolge der Elemente nicht an Speicheradressen gebunden ist, sind Probleme wie das Einfügen eines neuen Elements oder das Löschen eines Elements ausgesprochen elegant lösbar - bei einem Zeiger-Array müßten hier alle folgenden Einträge entsprechend verschoben werden.

Fangen wir mit dem Einfügen an: In die Liste



soll zwischen den Elementen B und C das neue Element B1 eingefügt werden. Grafisch dargestellt sieht die Sache wirklich recht einfach aus:

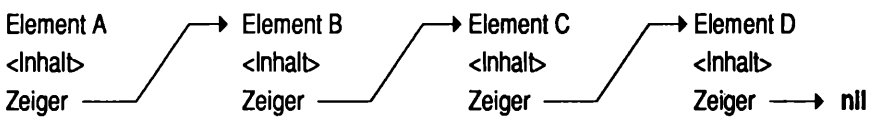


Hier wird also der Zeiger *Next* des Elements B - der vorher direkt auf C zeigt - auf die Adresse des neuen Elements B1 gesetzt und die Adresse von C im Feld *Next* von B1 gespeichert: Das Element C bleibt (wie alle folgenden Elemente) *unverändert.*

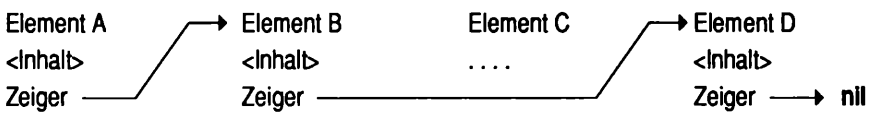
Die programmtechnische Realisierung dieser Operation ist erfreulicherweise auch nicht wesentlich komplizierter:

```
var
  LRoot : ListPtr;    { Zeiger auf den Start der Liste (A) }
  pLeft : ListPtr;    { Zeiger auf das linke Element (B) }
  pRight: ListPtr;    { Zeiger auf das rechte Element (C) }
  pNew: ListPtr;      { Zeiger für das neue Element (B1) }
  ...

  < Zeiger pLeft auf das Element B setzen >

  pRight := pLeft^.Next;  { pRight auf die Adresse von C }
  New(pNew);              { Erzeugen des neuen Elements B1 }
  pLeft^.Next := pNew;    { Zeiger "Next" in B auf B1 }
  pNew^.Next := pRight;   { Zeiger "Next" im neuen Element B1
                            auf das Element C }
(* GetMem(pNew^.Line, xx); *)
```

Das Löschen eines Elements ist noch etwas einfacher. Um aus der Liste



das Element C herauszunehmen, wird der durch C belegte Speicherplatz wieder freigegeben und das Feld *Next* von B direkt auf D gesetzt, also:



Mit Pascal-Anweisungen ausgedrückt:

```
....
< pLeft auf die Adresse von B >

pNew := pLeft^.Next;     { pNew auf die Adresse von C }
pRight := pNew^.Next;    { pRight auf die Adresse von D }

pLeft^.Next := pRight;   { Zeiger "Next" in B direkt auf die
                           Adresse von D }

(* with pNew do FreeMem(Line, Length(Line+1)); *)
  Dispose(pNew)          { Freigabe des von C belegten Speicherplatzes }
```

Das Programm *List1Demo* demonstriert diese beiden Operationen und kümmert sich zusätzlich um die beiden Spezialfälle, die bei dieser Besprechung noch nicht erwähnt wurden:

- Einfügen vor dem ersten Element oder Löschen des ersten Elements;
- Einfügen des ersten Elements in eine zuvor leere Liste.

Diese beiden Fälle sind die einzigen, bei denen der Zeiger auf den Start der Liste (*LRoot*) verändert werden muß.

List1Demo ist so einfach wie möglich aufgebaut: Das Programm kennt drei Befehle (Einfügen, Löschen, Quit) und belegt für jedes Element dieselbe Menge an Speicherplatz. Wenn Sie versuchen, an eine Liste mit 5 Elementen ein »Element 500« anzuhängen, verlängert *List1Demo* die Liste einfach um eins, d.h. hängt dieses Element als sechstes an.

```
program List1Demo;
{ Demonstration der Operationen "Einfügen" und "Löschen" in
  einer einfach verketteten Liste }
type
  ListPtr = ^ListElement;
  ListElement = record
                  Line : string[5];  (* für GetMem: ^string *)
                  Next : ListPtr;
                end;

var
  LRoot,                  { Start der Liste }
  pLeft,                  { Zeiger auf das "linke" Element }
  pRight,                 { Zeiger auf das "rechte" Element }
  p:      ListPtr;        { für die Ausgabe der Liste }

  NumElements: Integer;     { Anzahl der Listenelemente }
  QuitCmd, GotCmd: Boolean; { fürs Hauptprogramm }
  Ch: Char; x:Integer;      { dito }
```

```
function GetPrevious(Num: Integer): ListPtr;
{ Liefert die Adresse des Eintrags "Num-1" als Zeiger zurück. }
var p: ListPtr;
    x : Integer;
begin
  Dec(Num);
  if Num = 0 then GetPrevious := nil   { vor dem ersten Element }
  else begin
         p:= LRoot;
         for x:= 1 to Num-1 do    { für Element N: N-1 "Bewegungen" }
           p := p^.Next;
         GetPrevious := p;
       end;
end;
```

Für die Suche nach dem (nicht definierten) Vorgänger des ersten Listenelements liefert *GetPrevious* einen Zeiger zurück, der »nirgendwohin« zeigt.

```
procedure DeleteElement;
var p: ListPtr;
    x: Integer;
begin
  if LRoot = nil then
  begin
    Writeln('Liste ist leer.'); Exit;
  end;
  Write('Nummer (1..',NumElements,'): '); Readln(x);
  if x > NumElements then x := NumElements;
  if x < 1 then x := 1;   { Begrenzung }

  { Löschen des Elements }
  pLeft := GetPrevious(x);     { Adresse des vorhergehenden Elements }
  if pLeft = nil then          { Spezialfall 1: Anfang der Liste }
  begin
    pRight := LRoot^.Next;     { -> das zweite Element }
    Dispose(LRoot);            { Freigabe des ersten Elements }
    LRoot := pRight;           { LRoot auf das zweite Element }
  end
```

Die Löschung des momentan ersten Listenelements: *GetPrevious* liefert den Wert **nil** zurück, und *DeleteElement* greift direkt über *LRoot* auf den Zeiger *Next* dieses Elements zu. Eine spezielle Behandlung für den Fall, daß mit diesem ersten Element auch gleichzeitig das letzte Element gelöscht wird (d.h. die Liste hinterher leer ist), gibt es nicht: Das Feld *Next* des jeweils letzten Elements enthält den Wert **nil** zur Kennzeichnung des Listenendes - die Zuweisung *LRoot := pRight;* setzt *LRoot* in diesem Fall also automatisch auf **nil** zurück.

```
    else
    begin
      p:= pLeft^.Next;          { Zeiger auf das zu löschende Element }
      pRight := p^.Next;        { Zeiger aufs nächste Element (oder nil) }
      Dispose(p);               { Freigabe des Elements }
      pLeft^.Next := pRight;    { Schließen der Kette "am gelöschten }
     end;                       { Element vorbei" }

     Dec(NumElements);
  end;
```

Die Löschung eines beliebigen Listeneintrags: Das Feld *Next* des vorhergehenden Elements wird »um das zu löschende Element herum« auf dessen Nachfolger gesetzt. Eine spezielle Behandlung für den Fall, daß wir hier das letzte Element der Liste löschen, ist (wie oben) unnötig - *p^.Next* enthält in diesem Fall den Wert **nil**.

Wenn die Belegung für den Inhalt eines Listenelements über *GetMem* geschehen ist (d.h. das Feld *Line* als `^string` deklariert wurde), dann muß das zu löschende Element mit den folgenden Anweisungen wieder freigegeben werden:

```
(*          with p^ do
              FreeMem(Line, Length(Line^)+1;
            Dispose(p);
*)
```

Der Record, dessen Adresse in *p* gespeichert ist, belegt auch in diesem Fall eine fixe Anzahl von Bytes (4 für den Zeiger *Line*, 4 für den Zeiger *Next*, d.h. insgesamt 8) und kann deshalb über den typisierten Zeiger *p* freigegeben werden.

Ein **Hinweis:** Auch wenn die Reihenfolge dieser beiden Anweisungen technisch gesehen keine Rolle spielt (weil *Dispose* den Wert von *p* nicht verändert), sollte man sich trotzdem daran halten: definitionsgemäß betrachtet Pascal *p* danach als ungültig*).

```
procedure InsertElement;
var pNew: ListPtr;
    x: Integer;
begin
  Write('Nummer (1..',NumElements+1,'): ');  Readln(x);
  if x > NumElements + 1 then x:= NumElements + 1;  { Begrenzung }
  if x < 1 then x := 1;
  pLeft := GetPrevious(x);     { Zeiger auf das vorhergehende Element }

  New(pNew);                   { neues Element erzeugen }
```

*) ... und niemand weiß, was zukünftige Versionen des Compilers noch so alles bringen werden.

```
  { Einfügen des neuen Elements }
  if pLeft = nil then          { Spezialfall 1: Listenanfang }
  begin
    if LRoot = nil then        { Spezialfall 1_1: Liste ist leer }
    begin
      LRoot := pNew;           { in diesem Fall ist das neue Element }
      pNew^.Next := nil;       { sowohl das erste als auch das letzte }
    end
```

Im Gegensatz zum Löschen ist das Einfügen eines neuen Elements wesentlich komplexer als sein theoretisches Gegenstück. Der erste Spezialfall ist eine komplett leere Liste: zum einen wird *LRoot* direkt gesetzt (weil kein Vorgänger existiert), zum anderen müssen wir auch das Feld *Next* explizit setzen (weil kein Nachfolger vorhanden ist).

```
    else begin
     pRight := LRoot;          { sonst zeigt LRoot auf das alte erste }
     LRoot := pNew;            { Element: LRoot neu setzen und die }
     pNew^.Next := pRight;     { Adresse dieses Elements in pNew^ eintragen }
    end
  end
```

Spezialfall Nummer 2: Einfügen eines neuen Elements vor den Anfang der Liste. Einen Vorgänger haben wir hier naturgemäß nicht - weil aber der Fall »Liste leer« eigens behandelt wird, können wir uns darauf verlassen, daß ein Nachfolger existiert, dessen Adresse ohne weitere Prüfung in das Feld *Next* des neuen Elements eingetragen wird.

```
  else
  begin  { pLeft <> nil }
    pRight := pLeft^.Next;     { aus B->C wird B->B1->C. }
    pLeft^.Next := pNew;       { Kürzer wäre: pNew^.Next := pLeft^.Next; }
    pNew^.Next := pRight;      { und: pLeft^.Next := pNew; }
  end;
```

Der »Normalfall«: Das Feld *Next* des neuen Elements bekommt die Adresse seines Nachfolgers zugewiesen; im Feld *Next* des Vorgängers wird die Adresse des neuen Elements gespeichert. Eine gesonderte Behandlung für das *Anhängen* eines neuen Elements (d.h. für den Fall, daß kein Nachfolger existiert) ist unnötig: *pLeft^.Next* enthält dann den Wert **nil**, der wie jeder andere Zeiger in *pNew^.Next* kopiert wird.

```
  Inc(NumElements);
  Write('Inhalt: '); Readln(pNew^.Line);
end;
```

So gewagt der Aufruf von *Readln* auch aussieht - er ist völlig legal. Weil *pNew* ein typisierter Zeiger auf Records ist, deren Feld *Line* wir als `string[5]` deklariert haben, schneidet *Readln* Eingaben mit mehr als fünf Zeichen automatisch ab.

Wenn die Belegung für den Inhalt eines Listenelements über *GetMem* erfolgen soll (d.h. das Feld *Line* als `^string` deklariert ist), dann kommen wir allerdings nicht ohne eine temporäre Zwischenspeicherung aus:

```
Readln(WorkLine);   { WorkLine als String mit 255 Zeichen deklariert }
GetMem(pNew^.Line, Length(WorkLine)+1);  { Platz belegen, Line setzen }
pNew^.Line^ := WorkLine;  { Kopie in das Element }
```

Vorsicht! Ein Aufruf wie *Readln(pNew^.Line^)* hätte mehr als unerfreuliche Folgen: Weil der Zeiger *Line* zu diesem Zeitpunkt noch nicht definiert ist, schreibt *Readln* die gelesenen Zeichen irgendwohin. Eine Längenbegrenzung (hier wohl eher: Schadensbegrenzung) findet ebenfalls nicht statt, weil *Line* als Zeiger auf einen String mit maximal 255 Zeichen deklariert ist.

```
        { ------------------------------------- }
begin
  QuitCmd := False; NumElements := 0;
  LRoot := nil;

  repeat
    Write('(E)infügen (L)öschen (Q)uit: ');
    Readln(Ch); GotCmd := True;
    case UpCase(Ch) of
      'E': InsertElement;
      'L': DeleteElement;
      'Q': begin
             GotCmd := False;   { keine weitere Listenausgabe }
             QuitCmd := True;
           end;
       else
          GotCmd := False;
     end;  { case }

    if GotCmd then
    begin            { Ausgabe der Liste }
      p := LRoot; x:= 1;
      while p <> nil do
      begin
        Write(x,' ',p^.Line,'   ');   { "Nummer" und Inhalt }
        Inc(x);
        p := p^.Next;
      end;
      Writeln;
   end;  { if GotCmd }
  until QuitCmd;
end.
```

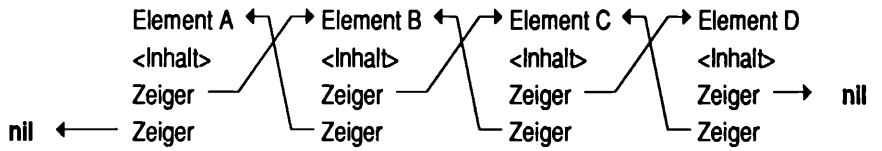
Das Hauptprogramm mit der Ausgabe des momentanen Listeninhalts: die Schleife endet, wenn als Zeiger auf das nächste Element der Wert **nil** gelesen wird, und arbeitet deshalb auch für leere Listen (*LRoot* = **nil**) korrekt.

Wenn mit *GetMem* gearbeitet wird (d.h. das Feld *Line* der Listenelemente als `^string` deklariert ist), dann muß *Write* den String übergeben bekommen, auf den *Line* zeigt, also

```
Write(x,' ',p^.Line^,'   '); { "Nummer" und Inhalt }
```

Doppelt verkettete Listen...

enthalten in jedem Element zwei Zeiger: einen auf den Vorgänger und einen zweiten auf ihren Nachfolger. Grafisch dargestellt:



Beim Löschen und Einfügen von Elementen muß hier also die doppelte Anzahl von Zeigern manipuliert werden - ein Aufwand, der sich in bestimmten Anwendungen aber mehr als bezahlt macht. Gegeben sei beispielsweise:

- eine Liste mit einer beliebigen Anzahl von Elementen, die jeweils zwischen 0 und 80 Zeichen speichern;
- ein Zeiger auf ihren Anfang (*LRoot*), ein Zeiger auf das letzte Element (*LLast*) sowie ein Zeiger auf ein beliebiges Element dieser Liste (*pNow*);
- und jemand, der sich in völlig irrationaler und unvorhersehbarer Folge relativ zum Element *pNow*^ in dieser Liste bewegt, vor oder nach *pNow*^ neue Elemente einfügt, manchmal das Element *pNow*^ löscht usw.

So ungefähr dürfte sich ein Benutzer wohl am besten darstellen lassen - jedenfalls aus der Sicht eines Textverarbeitungsprogramms bzw. -programmierers.

Bei einer Textverarbeitung finden praktisch alle Operationen relativ zum momentanen Punkt (sprich: dem Cursor) statt. Bereits ein vergleichsweise simples Problem wie die Bewegung des Cursors von Zeile 2510 nach Zeile 2509 läßt sich in einer einfachen Liste nur mit hohem zeitlichen Aufwand lösen:

```
p := LRoot;
for x:= 1 to 2508 do p := p^.Next;
```

Dasselbe gilt für jede Art von Veränderung, die über die »momentane Zeile« hinausgeht - und wie zuvor gezeigt, kommen wir ohne den jeweiligen Vorgänger weder beim Einfügen noch beim Löschen eines Elements aus.

Einfügen und Löschen

Um ein Element B1 in die Kette zwischen B und C einzufügen, wird wie bei einfachen Listen der Zeiger *Next* von B auf das neue Element gesetzt, und die Adresse von C im Zeiger *Next* des neuen Elements gespeichert:

```
B --> C     wird zu     B --> B1 --> C
```

Dasselbe geschieht bei doppelt verketteten Listen mit den Zeigern auf den jeweiligen Vorgänger:

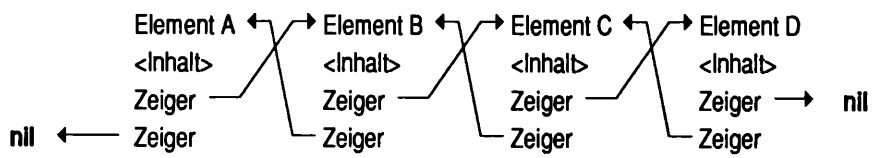
```
B <-- C     wird zu     B <-- B1 <-- C
```

Mit Pascal-Anweisungen ausgedrückt, wobei *pC* auf das Element zeigt, *vor* dem ein neues Element eingefügt werden soll (also dem Element »C« der Prinzipbeschreibung entspricht):

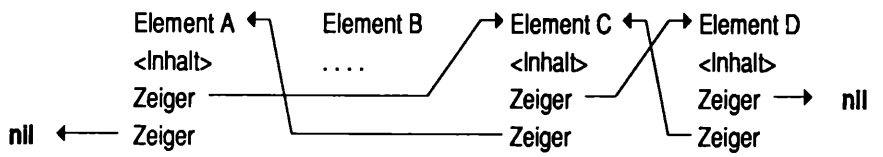
```
pLeft := pC^.Prev;        { Vorgänger }
pRight := pC;             { Cursorzeile = Nachfolger }
pNew^.Prev := pLeft;      { Vorgänger im neuen Element eintragen }
pNew^.Next := pRight;     { dito für Nachfolger }

{ "Einhängen" des neuen Elements }
pLeft^.Next := pNew;      { Zeiger "Next" im Vorgänger auf pNew }
pRight^.Prev := pNew;     { Zeiger "Prev" im Nachfolger auf pNew }
```

Das Löschen eines Elements ist auch hier etwas einfacher: Der Zeiger des Nachfolgers auf seinen Vorgänger wird auf den Vorgänger seines Vorgängers gesetzt, während der Zeiger des Vorgängers auf seinen Nachfolger die Adresse des Nachfolgers seines Nachfolgers zugewiesen bekommt.

Scherz beiseite. Aus



wird



Mit Pascal-Anweisungen ausgedrückt, wobei *pC* den Zeiger auf das zu löschende Element darstellt:

```
pLeft := pC^.Prev;    { Zeiger auf den Vorgänger }
pRight := pC^.Next;   { Zeiger auf den Nachfolger }
```

```
{ "Ausklinken" aus der Kette, Freigabe und Neusetzen von pC }
pLeft^.Next := pRight;    { A -> B -> C  ==>  A ---> C }
pRight^.Prev := pLeft;    { A <- B <- C  ==>  A <--- C }
Dispose(pC);              { Freigabe der Cursorzeile }
pC := pRight;             { pC auf die nächste Zeile(!) }
```

Achtung! Im Gegensatz zum Einfügen, wo der Cursor (*pC*) einfach auf dem Element stehenbleibt, *vor* dem eingefügt wird (was interessanterweise wie eine Vorwärtsbewegung aussieht), muß er nach dem Löschen einer Zeile explizit gesetzt werden - üblicherweise auf den Nachfolger.

»Dummy«-Elemente

Wie das Programm *List1Demo* demonstriert hat, müssen wir beim Einfügen eines neuen Elements zwischen drei Fällen unterscheiden:

- »Normales« Einfügen - das neue Element hat sowohl einen Vorgänger als auch einen Nachfolger, deren Zeiger entsprechend umgesetzt werden.
- Listenanfang - das neue Element wird vor dem ersten Element der Liste eingesetzt und hat keinen Vorgänger.
- Leere Liste - das neue Element hat weder einen Vorgänger noch einen Nachfolger.

Bei einer einfach verketteten Liste muß lediglich der Zeiger auf den Nachfolger umkopiert werden - ob dieser Zeiger auf ein weiteres Element zeigt oder den Wert **nil** hat, spielt für das Programm keine Rolle.

Bei einer doppelt verketteten Liste enthält der Nachfolger dagegen selbst einen Zeiger, dessen Wert verändert werden muß. Weil das Setzen eines »Nachfolgerzeigers« naturgemäß nur dann funktioniert, wenn auch ein Nachfolger vorhanden ist, müssen wir hier eine vierte Unterscheidung machen:

- Listenende - der Nachfolger ist **nil** und enthält keinen neu zu setzenden Zeiger auf einen Vorgänger.

Ähnliche Unterscheidungen sind für einen Cursor beliebiger Art (also einen Zeiger auf einen »momentanen« Record) notwendig - nämlich für den Fall, daß es diesen Record (noch) nicht gibt, weil die Liste leer ist oder sich der Cursor *hinter* dem letzten Element befindet.

Ein »Dummy«-Record, der grundsätzlich das Listenende darstellt, sich nicht löschen läßt und nie einen tatsächlichen Inhalt zugewiesen bekommt, löst diese Probleme ausgesprochen elegant:

- Normale Listenelemente haben auch dann einen Nachfolger, wenn sie am Ende der Liste stehen, außerdem ist die Liste nie völlig leer. Die Anzahl notwendiger Unterscheidungen für das Einfügen und Löschen reduziert sich auf zwei: »Anfang der Liste« und »in der Liste«.

- Wenn die Liste für den Benutzer als leer erscheint, zeigt der Cursor auf den »Dummy«-Record. Dasselbe gilt, wenn er sich hinter dem (vermeintlichen) Ende der Liste befindet.

LIST2DEMO

Das folgende Programm setzt die zuvor gezeigten Prinzipien in die Praxis um und konzentriert sich dabei (wie *List1Demo*) auf das Wesentliche*). Da hier mit einem Cursor gearbeitet wird, der immer auf die »momentane Zeile« zeigt, bin ich allerdings ohne einige Verzierungen und Schnörkel nicht ausgekommen: Der Listeninhalt wird nach jeder Operation in der rechten Hälfte des Bildschirms ausgegeben.

```
program List2Demo;
{ Demonstration einer doppelt verketteten Liste }

uses Crt;   { hauptsächlich für die Ausgabe der Listenelemente }

type
 List2Ptr = ^List2Element;
 List2Element = record
                  Line : string[20];
                  Prev : List2Ptr;    { Zeiger auf den Vorgänger }
                  Next : List2Ptr;    { Zeiger auf den Nachfolger }
                end;
var
  LRoot, LLast,           { Zeiger auf Beginn und Ende der Liste }
  pLeft, pRight,          { Vorgänger und Nachfolger }
  pCursor                 { Cursorposition }
               : List2Ptr;
  NumElements: Integer;
  QuitCmd: Boolean;
  Ch: Char;
```

Der Zeiger auf das Ende der Liste (*LLast*) zeigt immer auf den »Dummy«-Record und würde bei einem richtigen Programm für zwei Zwecke zugleich verwendet werden: den Sprung ans Textende und die Prüfung, ob jemand den »Dummy«-Record zu löschen versucht. *List2Demo* kennt nur diese Prüfung - ein Befehl für den Sprung ans Textende ist nicht definiert.

```
procedure ListList(CursorPos: List2Ptr);
{ Gibt den Listeninhalt (max. 24 Zeilen) in der rechten Hälfte
  des Bildschirms aus }
var p: List2Ptr;
    x,y : Integer;
```

*) *List2Demo* verwendet dasselbe Prinzip wie die Programme der Editor-Toolbox und läßt sich *selbstverständlich* leicht an eigene Zwecke anpassen und zu einem vollständigen Textverarbeitungssystem ausbauen. Dazu ist lediglich der Einbau von rund 130 Funktionen notwendig, was in etwa einer Verzwanzigfachung des Quelltextes entsprechen dürfte.

```
begin                          { Eine "typische Ausgabe":   }
  p := LRoot; x:= 1;
  while p <> nil do            {  1     Zeile 1             }
  begin                        {  2     eingefügte Zeile    }
    GotoXY(50,x);              { ==>    Zeile 2             }
    if p = CursorPos           {  4     noch eine Zeile     }
      then Write('==> ')       {    ..... }
      else Write(x:2,'  ');
    Write(p^.Line); ClrEol;
    Inc(x);
    p := p^.Next;
  end;
  for y:= x to 24 do     { Rest der rechten Bildschirmhälfte löschen }
  begin
    GotoXY(50,y); ClrEol;
  end;
end;
```

ListList wird nach jeder Operation aufgerufen und gibt daraufhin den gesamten Listeninhalt aus - unabhängig davon, ob sich überhaupt irgendetwas verändert hat. Da wir hier mit einem statischen Bildschirm arbeiten, ist die Anzahl der tatsächlichen Listenelemente auf 23 beschränkt.

```
procedure InsertElement(pC: List2Ptr);  { Cursor }
var
  pNew: List2Ptr;
begin
  New(pNew);     { neues Element erzeugen }

  if pC^.Prev = nil then  { oder: if pC = LRoot - Cursor am Anfang }
  begin
    LRoot^.Prev := pNew;      { Vorgänger ist jetzt vorhanden (= pNew) }
    LRoot := pNew;            { LRoot auf das nun erste Element der Liste }
    pNew^.Prev := nil;        { Vorgänger dieses Elements: keiner }
    pNew^.Next := pC;         { Nachfolger: das alte erste Element }
  end
```

Der erste (und einzige) Spezialfall: Wenn der Cursor auf das erste Listenelement zeigt, wird ein neues Element am Listenanfang eingefügt und *LRoot* entsprechend neu gesetzt.

```
  else
  begin
    pLeft := pC^.Prev;       { Vorgänger }
    pRight := pC;            { Cursorzeile = Nachfolger }
    pNew^.Prev := pLeft;     { Vorgänger im neuen Element eintragen }
    pNew^.Next := pRight;    { dito für Nachfolger }
```

```
      { "Einhängen" des neuen Elements }
      pLeft^.Next := pNew;     { Zeiger "Next" im Vorgänger auf pNew }
      pRight^.Prev := pNew;    { Zeiger "Prev" im Nachfolger auf pNew }
    end;
      { Eingabe des Inhalts }
    Writeln;
    Write('Inhalt: '); Readln(pNew^.Line);
    Inc(NumElements);
  end;
```

Der »Normalfall«, d.h. das Einfügen eines Elements an beliebiger Stelle: Weil wir uns hier immer vor dem tatsächlichen Listenende (d.h. dem »Dummy«-Record) befinden, sind keine weiteren Unterscheidungen notwendig. Die Zwischenspeicherungen in *pLeft* bzw. *pRight* sind nicht unbedingt erforderlich und dienen nur der Übersichtlichkeit - wenn Sie nichts gegen Monstren haben, können Sie diese beiden Variablen auch vollständig hinauswerfen:

```
(*   pNew^.Prev := pC^.Prev;  { alter Vorgänger von pC^ -> pNew }
     pNew^.Next := pC;        { Cursorzeile ist Nachfolger von pNew^ }
     pC^.Prev^.Next := pNew;  { pNew -> Next-Zeiger des Vorgängers }
     pC^.Prev := pNew;        { pNew -> pNew^.Next^.Prev ... }
*)
```

Wie in *List1Demo* müßte bei der Belegung von *Line* über *GetMem* eine temporäre Variable zur Zwischenspeicherung verwendet werden (also `Readln(WorkStr); / GetMem(pNew^.Line, Length(WorkStr)+1)); / pNew^.Line^ := WorkStr;`).

```
procedure DeleteElement(var pC: List2Ptr);  { Cursor }
begin
  if pC = LLast then  { Cursor auf dem Listenende }
  begin
    Writeln;
    Writeln('Letztes Element läßt sich nicht löschen');
    Write('Weiter mit RETURN: '); Readln;
    Exit;
  end;
```

DeleteEntry beginnt mit der Prüfung, ob sich der Cursor auf dem Listenende befindet, und meldet sich gegebenenfalls. Ein echtes Programm würde a) das Listenende nicht eigens auf dem Bildschirm darstellen und b) Löschbefehle einfach ignorieren, wenn sich der Cursor hinter dem Text (d.h. auf dem Listenende) befindet.

```
  if pC = LRoot then    { Löschung des ersten Elements }
  begin
    pC := pC^.Next;     { auf das nächste Element }
    pC^.Prev := nil;    { Zeiger "Prev" des nächsten Elements auf nil }
    Dispose(LRoot);     { Freigabe des ersten Elements }
    LRoot := pC;        { LRoot auf das zuvor zweite (und jetzt erste) El. }
  end
```

Der erste (und einzige) Spezialfall: die Löschung des ersten Listenelements. *LRoot* wird auf das zuvor zweite Element der Liste gesetzt; ein Vorgänger, dessen Feld *Next* verändert werden müßte, existiert nicht.
Achtung: *pC* ist als **var** deklariert - der »Cursor« wird hier also neu gesetzt.

```
    else
    begin    { zu löschendes Element ist nicht das erste }
      pLeft := pC^.Prev;   { Zeiger auf den Vorgänger }
      pRight := pC^.Next;  { Zeiger auf den Nachfolger }

      { "Ausklinken" aus der Kette, Freigabe und Neusetzen von pC }
      pLeft^.Next := pRight;   { A -> B -> C  ==>  A ---> C }
      pRight^.Prev := pLeft;   { A <- B <- C  ==>  A <--- C }
      Dispose(pC);             { Freigabe der Cursorzeile }
      pC := pRight;            { pC auf die nächste Zeile(!) }
    end;
      Dec(NumElements);
  end;
```

Der Normalfall: Das zu löschende Element hat sowohl einen Vorgänger als auch einen Nachfolger, deren Zeiger *Next* und *Prev* nun »an *pC^* vorbei« gesetzt werden. Auch hier ließe sich die Zwischenspeicherung mit *pLeft* und *pRight* auf Kosten der Übersichtlichkeit herausoptimieren:

```
(*      pC := pC^.Next;              { pC auf "C" }
        pC^.Prev := pC^.Prev^.Prev;  { A <---- C  }
        Dispose(pC^.Prev^.Next);     { Freigabe von "B" (via A.Next) }
        pC^.Prev^.Next := pC;        { A ----> C  }
*)
```

Wie im Programm *List1Demo* gezeigt, muß bei der Belegung von *Line* über *GetMem* zuerst *Line^* und dann *pC^* freigegeben werden (`FreeMem(pC^.Line, Length(pC^.Line^)+1))`; / `Dispose(pC);`).

```
        { ---------------------------- }
  begin
    New(LRoot);   { Dummy - "letztes Listenelement" }
    with LRoot^ do
    begin
      Next:= nil; Prev := nil;  { kein Nachfolger, kein Vorgänger }
      Line := ' --- Listenende --- ';
    end;
    LLast := LRoot;      { <== zeigt immer auf den "Dummy"-Record }
    pCursor := LRoot;    { Cursor auf den Listenanfang (= Dummy-Rec.) }
    NumElements := 1;
    QuitCmd := false;
```

```
  ClrScr;
  Writeln('Befehle: ');
  Writeln('   E - Cursor aufwärts');
  Writeln('   X - Cursor abwärts');
  Writeln('   I - Einfügen vor der Cursorzeile');
  Writeln('   Y - Löschen der Cursorzeile');
  Writeln('   Q - Ende');
  Writeln('-------------------------------------');
  ListList(pCursor);   { damit "--- Listenende ---" erscheint }
```

Der Beginn des Hauptprogramms mit der Erzeugung des »Dummy«-Records für das Listenende und der Ausgabe des »Befehlsmenüs«, das während des gesamten Programmlaufs unverändert auf dem Bildschirm stehenbleibt.

```
  repeat
    Window(1,9,40,25); ClrScr;   { löscht das "Eingabefenster" }
    Write('Befehl: ');
    Ch := UpCase(ReadKey);
    if Ch in ['E','X','I','Y','Q'] then
    begin
      Writeln(Ch);
      case Ch of
        'E': if pCursor^.Prev <> nil then pCursor := pCursor^.Prev;
        'X': if pCursor^.Next <> nil then pCursor := pCursor^.Next;
        'Y': DeleteElement(pCursor);
        'I': if NumElements < 24 then InsertElement(pCursor);
        'Q': QuitCmd := True;
      end;
      Window(1,1,80,25); ListList(pCursor);
    end;  { if Ch }
   until QuitCmd;
   GotoXY(1,24);
end.
```

Die tatsächliche Befehlseingabe findet hier in einem eigenen »Fenster« statt, das vor dem Aufruf von *ListList* wieder auf den gesamten Bildschirm erweitert werden muß.

Kapitel 4
Speicherresidente Programme und Interrupts

Herzlichen Glückwunsch, lieber Leser! Nach rund 230 Seiten sind wir nun wieder beim Programm *Nummer1* des Benutzerhandbuchs angekommen. Hier ist es noch einmal:

```
program Nummer1;
begin
  Writeln('Hallo!');
end.
```

Starten Sie Turbo Pascal (mit TURBO ↵) und geben Sie dieses Programm ein, wobei Sie auf die Semikolons am Ende der ersten und dritten Zeile achten sollten. Danach setzen Sie den Schalter *Destination* des Menüs *Compile* auf *Disk* und lassen das Programm mit einem Druck auf F9 compilieren. Beenden Sie TURBO mit ALT X und geben Sie auf der Kommandoebene von DOS den Befehl nummer1 ↵. Auf dem Bildschirm erscheint:

```
Hallo!
```

Soviel zu den humoristischen Einlagen in diesem Kapitel. Die Ausführung des Programms NUMMER1 läßt sich in drei Schritte unterteilen*):

1. Lokalisieren von NUMMER1.EXE auf der Diskette/Festplatte, Laden des Programms ab der »nächsten freien Adresse« in den Hauptspeicher, Relozierung von *far*-Aufrufen;
2. Sprung zum ersten Befehl von NUMMER1 und Ausführung;
3. Das Programm endet mit einem DOS-Aufruf und »wird aus dem Speicher entfernt«.

Betrachten wir diese Schritte einmal aus der Sicht des Betriebssystems:

1. Nachdem COMMAND.COM die Datei NUMMER1.EXE über diverse DOS-Aufrufe gefunden und die Länge dieser Datei bestimmt hat, folgt ein weiterer DOS-Aufruf, der NUMMER1.EXE ab »nächsten freien Adresse« in den Speicher lädt und startet.
   DOS »reserviert« einen Speicherblock für das Programm, d.h. speichert seine (absolute) Startadresse und seine Länge in einer Tabelle.

*) Der Gesamtablauf ist in Kapitel 1 in solcher Ausführlichkeit besprochen, daß ich mir hier die Unterschlagung einiger Einzelheiten leiste (wie z.B. der Unterteilung von COMMAND.COM in einen residenten und einen transienten Teil).

2. NUMMER1 wird ausgeführt. Da wir mit den Standardvorgaben für Stack und Heap arbeiten ({$M 16384, 0, 655360}), legt das Programm seine Fragmentliste auf der obersten physikalischen RAM-Adresse des Systems an.

3. NUMMER1.EXE endet mit dem Aufruf der DOS-Funktion $4C (»Terminate Process«). DOS gibt den reservierten Speicherbereich wieder frei (kürzer: löscht den Tabelleneintrag) und lädt COMMAND.COM ab der nächsten freien Adresse in den Speicher*). Da diese »nächste freie Adresse« mit der Startadresse von NUMMER1 zusammenfällt, wird unser Programm dabei überschrieben.

Die Funktion KEEP...

ist die zweite Möglichkeit, ein Programm zu beenden. Sie unterscheidet sich von »Terminate Process« hauptsächlich dadurch, daß der Tabelleneintrag nicht gelöscht, sondern lediglich neu gesetzt wird - DOS betrachtet den Speicherblock des Programms weiterhin als reserviert und lädt COMMAND.COM oberhalb dieses Blocks in den Hauptspeicher. Wegen dieses Ablaufs (»Terminate [but] stay resident« = Beenden, aber nicht aus dem Speicher entfernen) werden Programme, die mit einem Aufruf von KEEP enden, üblicherweise als *TSR-Programme* bezeichnet.

Eine kurze Demonstration:

```
program KeepDemo;

{ *** ACHTUNG: Dieses Programm darf NICHT von der integrierten
       Entwicklungsumgebung aus gestartet werden! }

{$M 1024,0,0}  { *** unbedingt notwendig! }
uses Dos;      { der Aufruf von KEEP ist im Unit DOS definiert }

var
  EndAddr: Word;

begin
  Writeln('Erstes Programmsegment: ', PrefixSeg);
  EndAddr := Word(Ptr(PrefixSeg,2)^);
  Writeln('Größe in Segmenten:     ', EndAddr - PrefixSeg);
  SetIntVec($00, SaveInt00);      { DIV0-Vektor zurück aufs Original }
  Keep(0); { Dageblieben! }
end.
```

*) Letzteres ist nur der Fall, wenn das Programm einige hundert dynamische Variablen belegt und in unzusammenhängender Folge wieder freigegeben hat (siehe Anhang D).

Wenn Sie dieses Programm mehrmals hintereinander starten, erhalten Sie eine Folge von Werten wie

```
Erstes Programmsegment: 9980
Größe in Segmenten:     305
Erstes Programmsegment: 10196
Größe in Segmenten:     305
Erstes Programmsegment: 10512
Größe in Segmenten:     305
```

Zwei Dinge sollten hier auffallen: Erstens berechnen wir den Start ab dem Programmsegment-Präfix (also nicht ab dem ersten tatsächlichen Befehl), zweitens liegen die Startadressen der Programme jeweils 306 Segmente auseinander. Tatsächlich wird der vom Programm belegte Speicherblock ab dem PSP gerechnet (und ist auch mit dieser Adresse in der DOS-Tabelle verzeichnet). Daß mehr Segmente belegt werden, als das Programm ermittelt, liegt an der dem PSP vorangestellten *Environment*-Tabelle (siehe Kapitel 1, Seite 4), die DOS sozusagen automatisch dazuschlägt.

Der Aufruf der DOS-Funktion KEEP am Ende von *KeepDemo* geschieht mit den folgenden Parametern:

```
AH = Funktionsnummer  ($31)
AL = Exit-Code        (Parameter der Pascal-Prozedur Keep)
DX = Anzahl der zu belegenden Segmente ($131 = 305)
```

Einen Parameter, welcher Speicherblock hier belegt bleiben soll, gibt es nicht - DOS »weiß« automatisch, daß es sich um den Block des zuletzt gestarteten Programms handelt.

Bevor wir uns der Frage zuwenden, wo der Sinn der diversen Aufrufe von *KeepDemo* liegt, noch eine Reihe von **lebenswichtigen Hinweisen:**

1. Programme, die mit einem Aufruf der DOS-Funktion KEEP enden, dürfen *nicht* von der integrierten Entwicklungsumgebung aus gestartet werden, weil sich TURBO.EXE in diesem Fall *unterhalb* des Programms im Speicher befindet. Das Ergebnis wäre ein reservierter Speicherblock inmitten des verfügbaren Speicherbereichs - eine Situation, der weder DOS noch TURBO gewachsen sind und die über kurz oder lang katastrophal endet.

   Wenn das Programm Interrupt-Vektoren setzt (siehe nächster Abschnitt) und/oder mit *Destination... Memory* übersetzt wurde, dann gibt es *sofort* Ärger, weil TURBO.EXE den Speicherbereich des Programms bei der nächsten Compilierung überschreibt - ein Aufruf der entsprechenden Interrupts führt danach ins Leere.
2. Ein Programm, das ohne Begrenzung von *HeapMax* compiliert wurde und mit KEEP endet, belegt den Hauptspeicher bis aufs letzte Bit und läßt keinen Platz

für das erneute Laden von COMMAND.COM. Anders gesagt: Wenn Sie bei Experimenten eine Meldung wie

```
Memory allocation error
Cannot load COMMAND - system halted
```

erhalten und zum RESET-Knopf greifen müssen - etwas anderes bleibt Ihnen in dieser Situation nicht übrig - dann haben Sie in Ihrem Programm den Compiler-Befehl `{$M xxx,yyy,zzz}` vergessen.

3. Der Wert des DX-Registers legt beim Aufruf von KEEP wirklich die Anzahl der Segmente ab dem PSP fest, die im Speicher bleiben sollen - einer der wenigen Punkte, zu denen Peter Norton im *Programmer's Guide* Unsinn geschrieben hat[1].

4. Der Initialisierungsteil von *System* belegt in der Version 4.0 drei Interruptvektoren (vgl. »Die Initialisierungsteile der Standard-Units« in Kapitel 1): $00 (für Divisionen durch 0), $23 (CONTROL-BREAK) und $24 (»Critical Error«). Zwei davon ($23 und $24) werden von DOS nach Ablauf eines Programms automatisch wieder auf ihre Originalwerte gesetzt[2] - unabhängig davon, ob der Rücksprung als »Terminate Process« oder via KEEP erfolgt. Um den dritten Vektor müssen wir uns »zu Fuß« kümmern: die Pascal-Standardprozedur *Keep* beschränkt sich auf die Ermittlung des Platzbedarfs und den abschließenden DOS-Aufruf - um eventuell veränderte Vektoren kümmert sie sich nicht im geringsten.

Nachdem nun alle Regeln beachtet wurden, befindet sich *KeepDemo* im Speicher - möglichst noch in mehrfacher Ausführung - und belegt dort Platz. Nicht nur, daß das Programm die Möglichkeiten des Computers in keiner Weise bereichert - die einzige Möglichkeit, *KeepDemo* wieder aus dem Speicher zu entfernen, besteht in einem Neustart des Systems.

Was man mit speicherresidenten Programmen alles anfangen kann, ist größtenteils Allgemeingut und das Thema des nächsten Abschnitts. Bis zur Beantwortung der Frage, wie man sich der gerufenen Geister wieder entledigt, wird es dagegen noch einige Seiten dauern.

Version 5.0

Nachdem die ersten beiden Kapitel dieses Buches rund 60 Detailunterschiede der beiden Versionen dokumentieren und TSR-Programme wirklich an die Grenzen einer Hochsprache gehen, sollte man eigentlich annehmen, daß die Medusa der Inkompatibilität gerade hier ihr gräßlich Haupt erhebt. Tatsächlich läßt sich dieses Untier aber mit *einem einzigen Befehl* erschlagen, der da lautet:

---

[1] Amerikanische Originalausgabe von 1985, Seite 302.

[2] Die Kehrseite der Medaille: das Abfangen von »Critical Errors« über {$I-} <I/O-Operation> {$I+} und einen darauffolgenden Aufruf von *IOResult* ist in TSR-Programmen nur über einen Trick möglich - wir kommen beim Programm PRTFILE weiter hinten in diesem Kapitel noch darauf zurück.

```
Writeln('Größe in Segmenten:     ', EndAddr - PrefixSeg);
{$IFDEF VER50}
  SetIntVec($3F, SaveInt3F); { V 5.0: Overlay-Vektor zurück }
{$ENDIF}
SetIntVec($00, SaveInt00);      { DIV0-Vektor zurück aufs Original }
Keep(0); { Dageblieben! }
```

Der Vektor $3F wird in der Version 5.0 von *System* auf eine definierte Abbruchroutine (»Overlay manager not installed«) gesetzt, damit der Aufruf von Overlay-Routinen vor der Aktivierung der dazugehörigen Verwaltung nicht ins Nirwana führt (vgl. »Der Initialisierungsteil von Overlay« in Kapitel 1).

Da dieser Vektor (im Gegensatz zu $00) nicht für einen bestimmten Zweck reserviert ist, käme man wahrscheinlich sogar ohne ein explizites Rücksetzen ungestraft davon.

Ein weiteres Detail - und bereits das letzte: KEEPDEMO zeigt in der Version 5.0 nicht jeweils 305, sondern nur 271 belegte Segmente an - und liefert damit einen weiteren Beweis dafür, wie gründlich Compiler und Laufzeitbibliotheken überarbeitet wurden[1].

interrupt-Prozeduren

Smiley1 demonstriert die erste Möglichkeit speicherresidenter Programme: den Ersatz einer Funktion durch eine andere.

Der Interrupt-Vektor 5 zeigt normalerweise auf die BIOS-Routine zur Ausgabe des Bildschirminhalts auf einen Drucker. *Smiley1* verändert diesen Vektor über den Aufruf von *SetIntVec* so, daß er auf die Pascal-Prozedur *Smiley* zeigt - nachdem das Programm gestartet worden ist, hat die Eingabe von SHIFT-PRTSC nicht mehr einen Bildschirmausdruck, sondern das Erscheinen des Zeichens `chr(1)` in der oberen linken Ecke des Bildschirms zur Folge[2].

Da der Aufruf von *Smiley* zu beliebigen Zeitpunkten geschehen kann und die Routine nicht auf die Werte der Prozessorregister zum Zeitpunkt der Unterbrechung zurückgreift, ist sie ohne Parameter deklariert. (Die Register werden aber auch in diesem Fall von Turbo Pascal auf dem Stack zwischengespeichert).

1) (Mes respects au maître de la cuisine).

2) Der Print Screen-Interrupt ist ideal zum Ausprobieren derartiger Programme, weil er a) nicht lebensnotwendig für das System und b) sehr einfach auszulösen ist.

```
program Smiley1;   { SMILEY1.PAS, 27/12/87 }
{ Ersetzt den Print Screen-Interrupt durch die Ausgabe
  des Zeichens chr(1) in der oberen linken Ecke des Bildschirms }

{$M 1024,0,0}    { **** notwendig!! ***** }

uses Dos;   { wg. SetIntVec, Keep }

procedure Smiley; interrupt; { ohne Parameter }
{ Diese Prozedur wird bei der Eingabe von Shift-PrtSc aufgerufen }
begin
  MemW[$B800:0] := $0701;  { für Hercules & Monochrom: MemW[$B000:0] }
end;

        { -------------------- }
begin
  {$IFDEF VER50}
    SetIntVec($3F, SaveInt3F); { V 5.0: Overlay-Vektor zurück }
  {$ENDIF}
  SetIntVec($00, SaveInt00);  { beide Versionen: DIV0-Vektor zurück }

  SetIntVec(5,@Smiley);  { INT $05 auf die Prozedur Smiley }
  Keep(0);
end.
```

Einige **Warnungen** und **Hinweise:**

- Die Deklaration der Routine *Smiley* als **interrupt** ist *lebensnotwendig* - wie im Kapitel 25 des Referenzhandbuchs beschrieben, erzeugt der Compiler hier zusätzliche Befehle, über die sämtliche Register des Prozessors zwischengespeichert und am Ende der Routine wiederhergestellt werden. Falls Sie diese Routine ohne den Zusatz **interrupt** deklarieren, erhalten Sie *keine Fehlermeldung* bei der Compilierung - aber mit nahezu hundertprozentiger Sicherheit einen Systemabsturz bei der Aktivierung über SHIFT-PRTSC.
- Bevor Sie die Routine *Smiley* nach eigenen Vorstellungen erweitern, lesen Sie bitte den übernächsten Abschnitt - Sie ersparen sich unter Umständen viel Ärger.

Ein besonders gelungenes Beispiel für gute Programmierung ist *Smiley1* nicht gerade. Unter anderem wird der Originalwert des Vektors $05 einfach überschrieben - die ursprüngliche Print Screen-Funktion ist also erst nach einem Neustart des Systems wieder verfügbar.

Lassen wir die diversen Ungereimtheiten einmal außer acht und kommen wir zu einer weiteren Möglichkeit: dem Einbau einer zusätzlichen Routine in zyklisch ablaufende Prozesse.

»Einhängen« zusätzlicher Funktionen

Die zweite Version von *Smiley* demonstriert das Umleiten eines Vektors auf eine Pascal-Prozedur, die zuerst den ursprünglichen Interrupt aufruft und danach eine eigene Funktion ausführt:

```
program Smiley2;   { SMILEY2.PAS, 27/12/87 }
{$M 1024,0,0}
uses Dos;

var
  SaveInt08: Pointer;   { für den Originalwert des Vektors $08 }

procedure Smiley; interrupt;
begin
  { Aufruf der Originalroutine. Über Intr wäre das etwas komplizierter,
    weil wir zuerst den Originalvektor wieder setzen und dann auch noch
    einen Dummy-Record (Typ Registers) übergeben müßten. }

    inline($9C/                    { PUSHF }
           $FF/$1E/SaveInt08);     { CALL FAR [SaveInt08] }

  MemW[$B800:0] := $0701;   { $B000:0 für Monochrom, Hercules }
end;

begin
  {$IFDEF VER50}
    SetIntVec($3F, SaveInt3F); { V 5.0: Overlay-Vektor zurück }
  {$ENDIF}
  SetIntVec($00, SaveInt00);   { beide Versionen: DIV0-Vektor zurück }

  GetIntVec(8,SaveInt08);   { Originalwert des Interrupt-Vektors $08 }
  SetIntVec(8,@Smiley);     { INT 08 auf die Routine Smiley }
  Keep(0);
end.
```

Der Befehl PUSHF vor dem Aufruf über den Originalvektor (*SaveInt08*) ist *unbedingt notwendig*: Die eigentliche Behandlungsroutine für den »Clock Tick« endet mit dem Befehl IRET, der nicht nur CS:IP, sondern auch die Flags wieder vom Stack holt.

Noch ein »unbedingt«: Der Aufruf der Originalroutine muß hier *auf jeden Fall* durchgeführt werden - wenn die eingebaute Uhr nicht in regelmäßigen Abständen vom Prozessor angesprochen wird, hängt das System. Für eigene Erweiterungen gilt dasselbe wie für *Smiley1* - bitte warten Sie damit noch einen Moment.

interrupt-Deklaration := Aprilscherz

Nachdem die beiden letzten Abschnitte demonstriert haben, wie einfach sich Interrupts durch Pascal-Prozeduren bearbeiten lassen, muß es nun gesagt werden: Vergessen Sie's. Für andere Zwecke als Männchen oben links im Bildschirm sind sie unbrauchbar. (Daß die Kritik an einem ansonsten wirklich hervorragenden Produkt wie Turbo Pascal so polemisch ausfällt, liegt nicht zuletzt daran, daß ich mich selbst einen guten Tag lang mit absolut unerklärlichen Fehlfunktionen herumgeärgert habe, bevor der Grund offensichtlich wurde - leider nach Drucklegung des Handbuchs*).

Machen wir die Erklärung so kurz wie möglich: Der vom Compiler eingesetzte Anfangs- und Endcode bei **interrupt**-Prozeduren speichert alle Register und setzt das Datensegment des Pascal-Programms. Was er nicht setzt, ist der Stack - und davon braucht Turbo Pascal (wie jede Hochsprache im Vergleich zu Assemblerprogrammen) geradezu atemberaubende Mengen.

COMMAND.COM reserviert beispielsweise rund 150 Bytes für den Stack - wenn dieses Programm von einem Turbo Pascal-Programm durch einen Interrupt unterbrochen wird und die entsprechende Turbo-Prozedur auch nur eine einzige Stringfunktion (Platzbedarf auf dem Stack: 256 Bytes, unabhängig der Stringlänge) benutzt, dann werden Bereiche außerhalb des Stacks von COMMAND.COM überschrieben: das System stürzt ab.

Noch eine Variation desselben Themas: Nach den Richtlinien von IBM und Microsoft sollte jedes Programm mindestens 16 Bytes Platz auf dem Stack für Unterbrechungen durch Interrupts zur Verfügung stellen - und die sind bei einer **interrupt**-Prozedur bereits durch die Speicherung der Prozessorregister belegt. Jeder Aufruf einer weiteren Pascal-Routine innerhalb der Behandlungsroutine, jede lokale Variable und jede Schleife verletzt diese Konvention und führt zu einem labilen System, das - abhängig von der Art des unterbrochenen Programms - plötzlich abstürzen kann.

Natürlich hat es einen Grund, warum der Stack bei **interrupt**-Prozeduren nicht automatisch gesetzt wird (und es auch nicht zu erwarten steht, daß eine neue Version des Compilers die folgenden Seiten überflüssig macht): Ein Umsetzen des Stacks ist nur dann möglich, wenn wir einige globale Variablen zu Hilfe nehmen. Anders gesagt: Die Speicherung von SS:SP auf dem Stack macht keinen Sinn, wenn wir danach diese beiden Register verändern (und damit die einzige Möglichkeit ausschalten, an die gespeicherten Werte später wieder heranzukommen).

---

*) Das Handbuch enthält auch in der Version 5.0 nicht mehr als den nebulösen Hinweis, daß »das unterbrochene (sic) Programm ausreichend Stackplatz zur Verfügung stellen muß«, weil a) die Version 5.0 in diesem Punkt leider hundertprozentig kompatibel ist und b) sich in der Dokumentation des Herstellers schlecht in Länge und Breite erklären läßt, wo derselbe Unsinn produziert hat.

INTFRAME.ASM...

entspricht im Prinzip dem Maschinencode, den Turbo Pascal an den Anfang und das Ende einer **interrupt**-Prozedur stellt - abgesehen davon, daß INTFRAME eben auch noch den Stack umsetzt. Die Behandlung eines Interrupts durch eine von INTFRAME »umgebene« Pascal-Prozedur läßt sich in vier Schritte unterteilen:

- Speicherung aller Prozessorregister in einer globalen Pascal-Variablen mit dem Namen *Regs*;
- Setzen des Datensegments und des Stacks von Turbo Pascal;
- Aufruf der Pascal-Prozedur *IntProc*. Diese Prozedur ist parameterlos und kann auf die Werte der Prozessor-Register über die Felder von *Regs* zugreifen. *IntProc* ist eine völlig normale Pascal-Prozedur, d.h. *nicht* als **interrupt** deklariert.
- Nach dem Ende von *IntProc*: Setzen der Prozessorregister mit den Feldern von *Regs*, Zurücksetzen von Datensegment und Stack auf die ursprünglichen Werte und Rücksprung zum unterbrochenen Programm mit IRET.

INTFRAME arbeitet mit einigen globalen bzw. statischen Variablen und ist deshalb *nicht reentrant*. Aus diesem Grund existiert die Flag-Variable *Active*: Weitere Aufrufe über den zu INTFRAME gehörigen Interrupt-Vektor werden ignoriert, solange die Pascal-Prozedur noch beschäftigt ist.

```
; INTFRAME.ASM - Rahmen zum Aufruf einer Pascal-Prozedur über
;                Interrupts. { 20/12/87 as}

Registers STRUC
rAX       dw ?          ; Struktur des Turbo Pascal-
rBX       dw ?          ; Datentyps "Registers" für
rCX       dw ?          ; den Assembler
rDX       dw ?
rBP       dw ?
rSI       dw ?
rDI       dw ?
rDS       dw ?
rES       dw ?
rFlags    dw ?
Registers ends
```

Diese Definition ist mit der Typ-Deklaration eines Records in Pascal vergleichbar: MASM erzeugt hier keine Datenbytes, sondern merkt sich nur die relativen Abstände der einzelnen Felder - das Symbol *rAX* bekommt also den Wert 0, *rBX* den Wert 2 usw. Das den Registernamen vorangestellte *r* dient nicht nur zur optischen Unterscheidung zu den Prozessorregistern: Weil MASM keine Unterschiede zwischen globalen Variablen, Feldnamen und Registernamen macht, endet der

Versuch einer Strukturdefinition mit Feldnamen wie AX und BX mit einer Fehlermeldung.

```
DATA    Segment Word PUBLIC
         EXTRN Regs:Word                    ; Registers-Record
         EXTRN TurboSS:Word,TurboSP:Word    ; SS:SP von Turbo

         EXTRN SSSave:Word,SPSave:Word      ; zum Speichern von SS:SP
         EXTRN CallCS:Word,CallIP:Word      ; zum Speichern von CS:IP
DATA    ends
```

Diese Variablen müssen innerhalb des Pascal-Programms global definiert sein. In *Regs* werden die Prozessorregister gespeichert, für die der Datentyp *Registers* entsprechende Felder definiert, *TurboSS* und *TurboSP* enthalten SS:SP von Turbo Pascal und müssen beim Start des Pascal-Programms entsprechend gesetzt werden.

Ein **Hinweis:** Erstaunlicherweise weist MASM den Versuch als fehlerhaft zurück, *Regs* mit dem zuvor definierten Typ *Registers* zu deklarieren - nach der Deklaration von *Regs* als *Word* haben Adressierungen der Form *Regs.AX* aber exakt denselben Effekt wie in Pascal (siehe weiter unten). Die Entwickler von Microsoft scheinen hier eine etwas andere Logik zugrundegelegt zu haben.

Die restlichen Variablen hätte man im Prinzip auch innerhalb von INTFRAME deklarieren können: *SSSave* und *SPSave* speichern den Stack des unterbrochenen Programms, *CallCS* und *CallIP* den Stand von CS:IP zum Zeitpunkt der Unterbrechung.

```
CODE    Segment Byte PUBLIC
        PUBLIC IntFrame
        EXTRN IntProc:near          ; die aufzurufende Pascal-Prozedur
        ASSUME cs:CODE, ds:DATA

Active  db 0                        ; (wird über CS adressiert)
IntFrame proc FAR
        push ax
        mov al,byte ptr Active      ; sind wir momentan bereits
        or al,al                    ; aktiv, d.h. wiederholter Interrupt?
        jz DoService                ; ... nein, Ausführung
        pop ax                      ; JA! Interrupt wird ignoriert
        iret
```

Der Beginn des CODE-Segments. Die Assembler-Routine *IntFrame* ist als PUBLIC deklariert, d.h. für das Pascal-Programm verfügbar. *IntProc* ist der Name der Pascal-Prozedur, die von *IntFrame* aus aufgerufen werden soll.

Achtung: Wir gehen hier und im weiteren davon aus, daß sich *IntFrame* und *IntProc* innerhalb *eines* Pascal-Programms befinden - wenn Sie *IntProc* im Implementationsteil eines Units deklarieren, müssen Sie mit *far*-Aufrufen arbeiten, d.h. die Zeile `EXTRN IntProc:near` in `EXTRN IntProc:far` umändern.

Die Flag-Variable *Active* ist innerhalb des CODE-Segments definiert und wird deshalb vom Assembler über das CS-Register adressiert. Auf diese Weise sparen wir uns das Setzen von DS für den Test auf »bereits aktiv«.

```
DoService: dec byte ptr Active   ; Flag auf $FF
        push ds
        mov ax,DATA              ; DS aufs Turbo-Datensegment
        mov ds,ax
        pop [Regs.rDS]           ; Stack (von oben nach unten): Flags
        pop [Regs.rAX]           ;          Rücksprungadresse   CS
        pop [CallIP]             ;                              IP
        pop [CallCS]             ;                              AX
        pop [Regs.rFlags]        ;                              DS
        mov [SSSave],ss          ; SS:SP sind um sechs
        mov [SPSave],sp          ; erhöht (wg. CS:IP, Flags)
        mov [Regs.rBX],bx
        mov [Regs.rCX],cx
        mov [Regs.rDX],dx        ; die restlichen Registerwerte
        mov [Regs.rBP],bp        ; in die Turbo-Variable REGS
        mov [Regs.rSI],si
        mov [Regs.rDI],di
        mov [Regs.rES],es
```

Der erste Teil des Startcodes: Die Konstante DATA wird in der .OBJ- bzw. .EXE-Datei als relozierbares Symbol gespeichert und erst beim Laden des Programms von DOS auf die Adresse gesetzt, ab der das Datensegment des Programms tatsächlich im Speicher steht.

Nachdem das DS-Register auf das Datensegment von Turbo Pascal gesetzt ist, können wir die Registerinhalte in *Regs* bzw. den restlichen Pascal-Variablen speichern. (Wie das Programm demonstriert, lassen sich Stack-Elemente nicht nur in Register zurückladen, sondern auch direkt in eine RAM-Speicherzelle übertragen).

```
        cli                      ; (eigentlich unnötig, s. Anhang A)
        mov ss,[TurboSS]         ; hier dürfen wir von niemandem
        mov sp,[TurboSP]         ; unterbrochen werden
        sti
```

Das sind die beiden Befehle, um die es eigentlich geht. Die Pascal-Variablen *TurboSS* und *TurboSP* müssen bei der Initialisierung des Programms (mit `TurboSP:= SPtr;` und `TurboSS:= SSeg;`) auf die entsprechenden Werte gesetzt werden.

Eine Unterbrechung (wie z.B. durch den »Clock Tick«) hätte an dieser Stelle unvorhersehbare Folgen*), deshalb werden Interrupts während dieser Operation unterdrückt.

*) Wenn SP im ursprünglichen Programm beispielsweise auf $FF00 innerhalb des Stacksegments zeigt und Turbo Pascal nur den Bereich SS:$0400..0000 für den eigenen Stack reserviert, dann überschreibt ein Interrupt *nach* dem Umsetzen von SS und *vor* dem Umsetzen von SP den Bereich <Turbo-Stack>:$FF00 - also Speicherzellen außerhalb des Programms.

```
        call IntProc            ; Aufruf der Turbo-Prozedur
```

Das Kernstück von INTFRAME: der Aufruf der Turbo-Prozedur. Ob er als *near* oder *far* durchgeführt wird, hängt von der Deklaration zu Beginn des CODE-Segments ab - in jedem Fall gilt aber, daß *IntProc* **nicht** als **interrupt** deklariert sein darf.

```
        mov es,[Regs.rES]
        mov bp,[Regs.rBP]
        mov di,[Regs.rDI]       ; Felder von REGS zurück in die
        mov si,[Regs.rSI]       ; Prozessor-Register
        mov dx,[Regs.rDX]
        mov cx,[Regs.rCX]
        mov bx,[Regs.rBX]
        mov ax,[Regs.rAX]       ; inklusive AX

        cli                     ; <-- dieses CLI ist auf jeden Fall notwendig
        mov ss,[SSSave]         ; SS:SP zurück aufs Original
        mov sp,[SPSave]
        sti
```

Nach der Ausführung von *IntProc* werden die Prozessorregister wieder mit den Feldern von *Regs* gesetzt, SS:SP bekommen ihre ursprünglichen Werte zurück.

```
        push[Regs.rFlags]       ; Flags und Rücksprungadresse auf
        push[CallCS]            ; den Stack des aufrufenden bzw.
        push[CallIP]            ; unterbrochenen Programms
        push[Regs.rDS]
        pop ds                  ; DS zurück aufs Original
        inc byte ptr Active     ; nicht mehr aktiv...
        iret
```

Die Flags und CS:IP des unterbrochenen Programms werden wieder auf den (Original-)Stack gebracht. Der Umweg über den Stack für das Setzen von DS ist unumgänglich - einen Befehl wie `MOV DS, DS:[Regs.rDS]` kennt der 8086 leider nicht.

Das Zurücksetzen des Flags *Active* geschieht wieder mit einer Adressierung über das CS-Register.

```
IntFrame endp                   ; Ende der Prozedur
CODE    ends                    ; Ende des Segments CODE
        end                     ; Ende des Quelltextes
```

Zu guter Letzt: die Formalitäten für MASM zum Abschluß der Prozedur *IntFrame*, des Segments CODE und des gesamten Quelltextes.

Ein Demonstrationsprogramm zu INTFRAME

Das Programm *IntFrameTest* hat exakt dasselbe Ergebnis wie *Smiley1* - programmiertechnisch bestehen allerdings zwei Unterschiede:

- Die beim Interrupt aufgerufene Pascal-Prozedur hat den (durch INTFRAME.ASM vorgegebenen) Namen *IntProc* und ist - ich wiederhole es noch einmal - *nicht* als **interrupt** deklariert.
- *IntProc* ist beliebig durch Aufrufe anderer Pascal-Prozeduren, lokale Variablen, Stringfunktionen usw. erweiterbar.

Allerdings kommen wir auch hier (noch) nicht um eine **Warnung** herum: Die von *IntProc* ausgeführten Operationen dürfen keine Aufrufe von DOS-Funktionen enthalten*). (Zu dieser Regel gibt es allerdings eine Ausnahme, die das Programm TRACE86 im nächsten Abschnitt demonstriert: Wenn Sie das Unit *Crt* in Ihr Programm aufnehmen, arbeiten *Write* bzw. *Writeln* ohne DOS-Aufrufe und sind deshalb verwendbar).

```
program IntFrameTest;  { INTFTEST.PAS - 20/12/87 as }
{ Demonstration des Aufrufs einer Turbo Pascal-Prozedur durch einen
  Interrupt über IntFrame.ASM - hier mit Shift-PrtSc }

{$M 1024,0,0}              { HeapMax = 0 - UNBEDINGT NOTWENDIG }

uses Dos;    { wg. Registers, SetIntVec usw. }

var
  Regs: Registers;
  SSSave, SPSave: Word;    { zum Speichern von SS:SP }
  TurboSS, TurboSP: Word;  { SS:SP von Turbo }
  CallCS,CallIP: Word;     { hier finden sich CS:IP des
                             aufrufenden Programms }

procedure IntFrame; external;
{$L IntFrame}

{ Die Prozedur IntProc wird von der Routine IntFrame beim Interrupt
  aufgerufen, die Prozessor-Register befinden sich in der globalen
  Variablen Regs. CallCS und CallIP dürfen nicht verändert werden! }

procedure IntProc;
begin
  Mem[$B800:0] := 1;   { $B000:0 für Monochrom, Hercules }
end;
```

*) Zum Warum, Wieso und vor allem »Wann doch«: siehe »DOS über die Schulter geschaut« weiter hinten in diesem Kapitel.

```
begin
  TurboSS := SSeg; TurboSP := SPtr;   { *** UNBEDINGT NOTWENDIG *** }
  {$IFDEF VER50}
    SetIntVec($3F, SaveInt3F);   { V 5.0: Overlay-Vektor zurück }
  {$ENDIF}
  SetIntVec($00, SaveInt00);     { beide Versionen: DIV0-Vektor zurück }

  SetIntVec(5,@IntFrame);        { Shift-PrtSc-Interrupt belegen }
  Keep(0);  { TSR, Exit-Code 0 }
end.
```

Eine echte Anwendung: TRACE86

Das folgende Programm arbeitet nach demselben Prinzip wie *Smiley2*, hängt sich also in den »Clock Tick« ein. Im Unterschied zu *Smiley2* erfreut es den Benutzer aber nicht mit einer Spielkarte, sondern mit dem Stand von CS:IP des jeweils unterbrochenen Programms. (Die Idee dazu stammt von einem Herrn namens Chuck Fricano aus Pittsburgh, dessen Maschinenprogramm TRACE02 seit einigen Jahren als *Public Domain* in Umlauf ist und mir bei der Suche nach komplizierteren Fehlern schon einige Dienste geleistet hat).

Wie im letzten Abschnitt angedeutet, müssen wir auch hier ohne DOS-Aufrufe auskommen - wenn wir das Unit *Crt* mit in das Programm aufnehmen, ist die Verwendung von *Writeln* aber erlaubt, weil *Crt.Writeln* direkt in den Bildspeicher schreibt bzw. einen BIOS-Aufruf dazu benutzt.

```
program TRACE86;          { 20/12/87 as }

{ Zweite Demonstration von INTFRAME.ASM: Zyklischer Aufruf einer
  Pascal-Prozedur durch Einhängen in den "Clock Tick" und Ausgabe
  des Standes von CS:IP im Halbsekundentakt. }

{$M 1024,0,0}              { HeapMax = 0 - UNBEDINGT NOTWENDIG }

uses Crt,     { für Write, GotoXY }
     Dos;     { wg. Registers, SetIntVec usw. }
var
  SaveInt08: Pointer;      { zum Speichern des Originalvektors }
  Regs: Registers;
  SSSave, SPSave: Word;    { zum Speichern von SS:SP }
  TurboSS, TurboSP: Word;  { SS:SP von Turbo }
  CallCS,CallIP: Word;     { CS:IP des aufrufenden Programms }
  CountDown,               { Zähler für die Anzahl der Interrupts }
  CountStart: Integer;

procedure IntFrame; external;    {$L INTFRAME}
```

Dieser Teil unterscheidet sich von *IntFrameTest* nur durch die Aufnahme des Units *Crt* sowie die Deklaration der Variablen *CountDown* und *CountStart*:

CountStart legt den Anfangswert von *CountDown* fest und sollte einen Wert zwischen 1 (Ausgabe 18 Mal in der Sekunde) und 18 (eine Ausgabe pro Sekunde) haben. *CountDown* wird durch jeden »Clock Tick« um eins erniedrigt - wenn der Wert 0 erreicht ist, folgt die Ausgabe von CS:IP, danach wird *CountDown* wieder auf den Wert von *CountStart* gesetzt.

```
procedure HexWordPrint(w:Word);
{ Wird von IntProc aufgerufen und gibt den Wert von w in
  hexadezimaler Form (ggf. mit führenden Nullen)
  ab der momentanen Position des Cursors aus. }
var
  x, Chx: Integer;  { Zähler, "Char" }
begin
  for x:= 1 to 4 do  { 4 Stellen }
  begin
    Chx := (w and $F000) shr 12;  { das jeweils oberste Digit }
    w := w shl 4;     { w um eine Hex-Stelle nach links }
    if Chx < 10 then
      Write(chr(Chx + Ord('0')))            { '0'..'9' }
      else Write(chr(Chx - 10 + Ord('A'))); { 'A'..'F' }
  end;
end;

procedure IntProc;
{ Diese Prozedur wird durch den "Clock Tick" aufgerufen }
var
  CursorX, CursorY: Byte;  { für die Originalposition des Cursors }
begin
  { Aufruf der Originalroutine }
    inline($9C/                  { PUSHF }
         $FF/$1E/SaveInt08);   { CALL FAR [SaveInt08] }

  Dec(CountDown);             { Zähler }
  if CountDown <= 0 then
  begin
    CursorX := WhereX; CursorY := WhereY;  { Originalposition des Cursors }
    GotoXY(71,1);  { Cursor in die obere rechte Ecke }
    HexWordPrint(CallCS);   { CS-Register des unterbrochenen Programms }
    Write(':');
    HexWordPrint(CallIP);   { IP-Register des unterbrochenen Programms }
    GotoXY(CursorX,CursorY);  { Cursor zurück }
    CountDown := CountStart;  { Zähler wieder auf den Startwert }
  end;
end;
```

Der Aufruf der Originalroutine für den »Clock Tick« findet in jedem Fall statt - die Ausgabe des Standes von CS:IP dagegen nur, wenn *CountDown* entsprechend heruntergezählt ist.

```
begin
  TurboSS := SSeg;
  TurboSP := SPtr-$40;   { *** UNBEDINGT NOTWENDIG *** }

  CountStart := 9;  { zwei Ausgaben pro Sekunde }
  CountDown := 0;   { erste Ausgabe sofort }

  GetIntVec(8,SaveInt08);          { Speichern des Originalvektors }
  CheckBreak := False;

  {$IFDEF VER50}
    SetIntVec($3F, SaveInt3F); { V 5.0: Overlay-Vektor zurück }
  {$ENDIF}
  SetIntVec($00, SaveInt00);   { beide Versionen: DIV0-Vektor zurück }
  SetIntVec($1B,SaveInt1B);    { ^Break-Interrupt von CRT zurück }

  { *** Achtung: Diese Anweisung muß die letzte vor KEEP sein! }
  SetIntVec(8,@IntFrame);          { Setzen von INT 08 auf unsere Routine }

  Keep(0);  { TSR, Exit-Code 0 }
end.
```

Das Programm läßt sich recht einfach um eine Auswertung von *ParamStr(1)* erweitern, mit deren Ergebnis dann *CountStart* gesetzt wird - ein Feature, das hier der Übersichtlichkeit halber fehlt.

Das Setzen des Interrupt-Vektors $08 ist eine reichlich kritische Angelegenheit: Ab diesem Zeitpunkt kann das Programm durch einen »Clock Tick« aufgerufen werden und unterbricht sich damit sozusagen selbst. Wenn wir die obersten $40 Bytes des Stacks nicht reservieren würden, sähen die Schritte in die unvermeidliche Katastrophe ungefähr so aus:

- Wir haben 1024 Bytes für den Stack reserviert, SP zeigt innerhalb des Stacksegments also auf die Adresse $400.
- Aufruf von *Keep*: Die Laufzeitbibliothek erzeugt einen Record des Typs *Registers*, setzt ihn mit den entsprechenden Werten und packt ihn auf den Stack, belegt also den Bereich $3E8..$3FF innerhalb des Stacksegments.
- Aufruf von *MsDos* durch *Keep*: *MsDos* setzt *IntNo* = $21, *Regs.AH* = $31 und ruft *Intr* auf. Hier wird der übergebene *Registers*-Record ein weiteres Mal auf den Stack gebracht und mit diversen POP-Operationen in die einzelnen Prozessorregister kopiert, bevor schließlich der eigentliche Interrupt-Aufruf erfolgt. Bis zu diesem Zeitpunkt sind für Parameter, Rücksprungadressen usw. rund $30 Bytes auf dem Stack belegt (SS:$3D0..$3FF).

Irgendwo inmitten dieser Operationen ist der nächste »Clock Tick« fällig: Über INTFRAME werden SS:SP auf $400 innerhalb des Turbo-Stacks gesetzt, lokale Variablen belegt und Prozeduren aufgerufen, d.h. Speicherzellen von der Adresse $400 abwärts innerhalb des Stacksegments überschrieben. Weil es sich dabei um dieselben Speicherzellen handelt, in denen der momentan laufende Aufruf von *Keep* seine Parameter und Rücksprungadressen gespeichert hat, geht es nach der Bearbeitung des Interrupts innerhalb von *Keep* bestenfalls noch in die ewigen Jagdgründe weiter.

Hilfe für den Zauberlehrling: Entfernbare TSR-Programme

Egal, in welcher der gängigen Computerzeitschriften man etwas zum Thema »speicherresidente Programme« sucht - am Ende des jeweiligen Artikels steht so gut wie immer der verschämte Hinweis, daß sich das Programm wohl mit einem Neustart des Computers am besten wieder entfernen läßt. Das im letzten Abschnitt vorgestellte Beispiel TRACE86 (und übrigens auch sein Original TRACE02) haben allesamt dieselbe Krankheit: Nicht nur, daß man Sie mit CTRL-ALT-DEL (oder massiveren Maßnahmen) wieder entfernen muß - sie merken auch nicht, ob sie bereits geladen sind. Anders gesagt: Ein mehrfacher Start von TRACE86 stapelt immer weitere Versionen des Programms im Hauptspeicher übereinander.

Dabei ist es gar nicht so schwierig, die einmal beschworenen Dämonen wieder loszuwerden - es bedarf lediglich eines gewissen Überbaus. Ein wieder aus dem Speicher entfernbares Programm muß:

- den Originalwert des veränderten Interrupt-Vektors an einer Stelle speichern, die in irgendeiner Form auch für andere Programme zugänglich ist;
- eine Prüfung enthalten, ob sich bereits eine Version seiner selbst im Speicher befindet;
- einen Kommandozeilen-Parameter (z.B. »OFF«) erkennen. Wenn dieser Parameter angegeben ist und sich bereits eine Version des Programms im Speicher befindet, dann wird der Originalvektor wieder gesetzt und die zuvor eingelagerte Version über die Freigabe ihres Speicherblocks (DOS-Funktion $49) hinausgeworfen.
- Die letzte Forderung ergibt sich aus der ersten und der dritten: Das Programm endet seinerseits nur dann mit einem Aufruf von *Keep*, wenn a) sich nicht bereits eine Version davon im Speicher befindet und b) nicht der Kommandozeilen-Parameter »OFF« angegeben wurde. In allen anderen Fällen endet es mit dem DOS-Aufruf »Terminate Process«, d.h. auf normale Weise.

Nachdem wir mittlerweile bei verwendbaren TSR-Programmen angelangt sind, müssen wir noch eine weitere Unterscheidung einführen - und zwar zwischen *funktionsgesteuerten Software-Interrupts* und *Hardware-Interrupts*. Hardware-Interrupts wie der »Clock Tick« können ein anderes Programm zu beliebigen Zeitpunkten unterbrechen und dürfen deshalb keine Register verändern bzw. Ergebnisse zurückliefern, funktionsgesteuerte Software-Interrupts werden dagegen von einem Programm zu definierten Zeitpunkten (und normalerweise mit einer Funktionsnummer im AH-Register) aufgerufen. In den meisten Fällen erwartet das aufrufende Programm Ergebnisse in Form veränderter Registerinhalte.

Funktionsgesteuerte Software-Interrupts

Das Programm *RemoveableFunctionInterrupt* verwendet den (ansonsten unbenutzten) Interrupt-Vektor $90 und arbeitet wiederum mit einem Aufruf der Pascal-Prozedur *IntProc* über INTFRAME.ASM. *IntProc* wird hier allerdings nicht zu beliebigen Zeitpunkten aufgerufen (wie über die Eingabe von SHIFT-PRTSC), sondern »gezielt« innerhalb eines Programms:

```
MOV AH,Funktionsnummer
INT $90
```

Die Funktionsnummern im Bereich von $FD..$FF sind für die Installation bzw. das Entfernen einer installierten Version reserviert und dürfen im »Normalbetrieb« nicht verwendet werden:

$FD — *IntProc* liefert drei Kennbytes in den Registern AX, BX und CX zurück. Über diese Kennbytes kann *RemoveableFunctionInterrupt* vor dem Zurücksetzen des Interrupt-Vektors feststellen, ob es sich bei einem bereits eingelagerten Programm auch wirklich um eine Version seiner selbst handelt.

$FE — Das »Funktionsergebnis« von *IntProc* ist der Original-Interruptvektor in den Registern DX und AX.

$FF — *IntProc* liefert die Segmentadresse seines Programmsegment-Präfix als Ergebnis. Diesen Wert brauchen wir, um den von einem eingelagerten Programm belegten Speicherblock wieder freizugeben (DOS-Funktion $49).

Der Aufruf über den Interruptvektor $90 mit einer Funktionsnummer im Bereich von $00..$FC hat bei diesem einfachen Beispiel durchgehend dasselbe Ergebnis, nämlich die schon bekannte Ausgabe des Zeichens `chr(1)` in der oberen linken Ecke des Bildschirms.

```
program RemoveableFunctionInterrupt;  { REMFNINT.PAS, 20/12/87 as }
{ Demonstration eines speicherresidenten Programms, das mit Funktions-
  codes arbeitet und sich auch wieder aus dem Speicher entfernen läßt. }
{$M 2000,0,0}              { HeapMax = 0 - UNBEDINGT NOTWENDIG }
{$S+,R-,I-}
uses Dos;    { wg. Registers, SetIntVec usw. }

type
  PWord = record   { Typ-Umwandlung für Zeigerwerte }
     Off: Word;       { Offset }
     Seg: Word;       { Segment }
   end;

const
  NEWINT = $90;   { Nummer des zu setzenden Interrupts }
  ID1 = 20;       { Kennbytes für die ID-Funktion - am einfachsten }
  ID2 = 12;       { läßt sich dazu das Erstellungsdatum des }
  ID3 = 87;       { Programms verwenden. }
  OFFCMD: string[3] = 'OFF'; { Kommandozeilen-Parameter }

var
  CmdStr: string[3];
  Off: Boolean;            { Flag für "Parameter OFF" }
  Installed: Boolean;      { Flag für "bereits installiert" }
  Ch: Char; x:Integer;
  EnvSeg: Word;            { Environment-Segmentadresse }
  SaveIntVec: Pointer;     { zum Speichern des Originalvektors }
  p: Pointer;
  Regs: Registers;         { zur Parameterübergabe }
  SSSave, SPSave: Word;    { zum Speichern von SS:SP }
  TurboSS, TurboSP: Word;  { SS:SP von Turbo }
  CallCS,CallIP: Word;     { hier finden sich CS:IP des
                             aufrufenden Programms }

procedure IntFrame; external;  { der "Rahmen" für den Aufruf von IntProc }
{$L INTFRAME}
```

Bis auf die Deklaration der Konstanten ID1..ID3 und die (in Kapitel 3 besprochene) Typdefinition *PWord* bietet dieser Teil nichts wesentlich Neues.

```
procedure IntProc;
begin
  with Regs do
  begin
    if AH > $FC then
    case AH of
      $FD: begin  { Kennbytes - hier das Erstellungsdatum des Programms }
             AX := ID1; BX := ID2; CX:= ID3;
           end;
```

```
        $FE: begin { Zurückliefern des Originalvektors }
               AX := PWord(SaveIntVec).Off; ES := PWord(SaveIntVec).Seg;
             end;
        $FF: begin { Zurückliefern des Programmsegment-Präfix }
               ES := PrefixSeg;
             end;
      end { case }
     else begin    { Die eigentlichen Funktionen von IntProc,
                     ggf. mit weiteren Prüfungen von AH }

             Mem[$B800:0] := 1;   { $B000:0 für Monochrom, Hercules }
          end;
    end; { with Regs }
  end; { der Routine }
```

In diesem Beispiel ist die eigentliche Funktion von *IntProc* wohl der kürzeste Teil - hier könnten sich selbstverständlich weitere Prüfungen des AH-Registers anschließen.

```
  begin    { ** Hauptprogramm ** }

    { 1. Schritt: Kommandozeilen-Parameter "Off"? }
    if ParamCount >= 1 then
    begin
      Off := True;
      CmdStr:= Copy(ParamStr(1),1,3);
      for x:= 1 to 3 do
        if UpCase(CmdStr[x]) <> OFFCMD[x] then Off:= False;
      if not Off then
      begin
        Writeln('Ungültiger Kommandozeilen-Parameter: ', ParamStr(1));
        Halt(1);
       end;
     end
    else Off := False;
```

Der erste Schritt innerhalb des Hauptprogramms mit der Prüfung, ob der Kommandozeilen-Parameter OFF angegeben wurde.

```
    { 2. Schritt: Prüfung, ob das Programm bereits installiert wurde.
      Wenn ja:  a) Kommandozeilen-Parameter "Off" -> hinauswerfen
                b) ansonsten Meldung "bereits installiert" und Abbruch }
    Installed := False;
    GetIntVec(NEWINT,SaveIntVec);
    if (PWord(SaveIntVec).Off = Ofs(IntFrame))
       and (SaveIntVec <> nil) then
```

```
begin
  with Regs do
  begin
    AH := $FD;  { Funktionscode: "ID" }
    BX := 0; CX := 0;
    Intr(NEWINT,Regs);
    Installed := (AX = ID1) and (BX = ID2) and (CX = ID3);
  end;
end;
```

Wenn der Offset-Anteil des Interrupt-Vektors denselben Wert wie die Adresse von *IntFrame* in dieser Version hat, dann können wir eigentlich davon ausgehen, daß sich bereits eine Version des Programms im Speicher befindet. Die Prüfung, ob ein Aufruf dieser Version mit dem Funktionscode $FD die erwarteten Kennbytes liefert, schließt mögliche Verwechslungen mit anderen Programmen aus, deren Interrupt-Routine zufällig denselben Adress-Offset hat. (Die Wahrscheinlichkeit für gleiche Offset-Anteile verschiedener Interrupt-Programme dürfte bei rund 1:60000 liegen - daß das entsprechende Programm aber bei einem Aufruf mit AH = $FC dieselben Kennbytes zurückliefert, dürfte rechnerisch ähnlich ausgeschlossen sein wie Unfälle bei Atomreaktoren. Oder auch: Garantieren kann man so etwas natürlich nicht*).

```
if (not Installed) and Off then
begin
  Writeln('Programm war nicht installiert.');
  Halt(1);
end;

if Installed then
begin
  if not Off then
  begin
    Writeln('Programm ist bereits installiert.');
    Halt(0);
  end
  else begin
    with Regs do
    begin
      AH := $FE; { Funktionscode: Original-Interrupt zurückliefern }
      Intr(NEWINT,Regs);
      PWord(SaveIntVec).Seg := ES; PWord(SaveIntVec).Off := AX;
      AH := $FF; { Funktionscode: PSP-Segment in ES zurückliefern }
      Intr(NEWINT,Regs);
      EnvSeg := Word(Ptr(ES,$2C)^);
```

*) Eigentlich ist das nicht so witzig.

```
        AH := $49;        { DOS-Funktion: Free Memory (Segment in ES }
        Intr($21,Regs);   { Freigabe des Programms }
        ES := EnvSeg;
        AH := $49;
        Intr($21,Regs);   { Freigabe des Environments }

        { Und schließlich: Zurücksetzen des Interrupts }
        SetIntVec(NEWINT,SaveIntVec);
      end;  { with Regs }
    Writeln('Programm aus dem Speicher entfernt');
    Halt(0);
  end; { else }
end; { if Installed }
```

Wenn der Kommandozeilen-Parameter OFF angegeben wurde und das Programm eine installierte Version seiner selbst erkannt hat, dann folgen sechs Schritte, um diese Version aus dem Speicher zu entfernen:

1. Ermittlung des ursprünglichen Interrupt-Vektors über den Funktionsaufruf $FE und Speicherung in der Variablen *SaveIntVec*.
2. Ermittlung des PSP-Segments des installierten Programms über den Funktionsaufruf $FF. Die Speicherung dieser Adresse in *Regs.ES* geschieht direkt durch den Aufruf.
3. Die Bytes $2C/2D im PSP-Segment des installierten Programms enthalten die Segmentadresse der zum Programm gehörigen *Environment*-Tabelle (vgl. Kapitel 1). Diese Segmentadresse wird in der Variablen *EnvSeg* gespeichert.
4. Der Aufruf der DOS-Funktion »Free Allocated Memory« (Funktionscode $49) mit der Adresse des PSP in *Regs.ES* gibt den Speicherblock wieder frei, der dem eingelagerten Programm und seinem PSP zugeordnet ist.
5. Der Aufruf derselben DOS-Funktion mit der Adresse der *Environment*-Tabelle in *Regs.ES* gibt auch diesen Speicherbereich wieder frei. Damit ist das eingelagerte Programm komplett aus dem Speicher »entfernt«. (Wirkliche Veränderungen innerhalb dieses Bereiches finden nicht statt - erst wenn auch das momentan laufende Programm endet, wird der vorher freigehaltene Speicherbereich durch das Nachladen von COMMAND.COM überschrieben).
6. Der Interrupt-Vektor wird wieder auf seinen ursprünglichen Wert gesetzt, danach endet das laufende Programm.

(Die Unart, das Programm gleich an vier verschiedenen Stellen enden zu lassen, wird in einer folgenden Version beseitigt, die die Prüfung auf »bereits installiert« als Funktion definiert - solange wir diese Schritte im Hauptprogramm stattfinden lassen, ist mir eine größere Zahl »eingestreuter« *Halt*-Befehle aber lieber als eine endlos verschachtelte Kette von **if..then**-Anweisungen).

```
  { 3. Schritt (falls das Programm bis hierher kommt): }
  {    Die eigentliche Installation.                   }

  TurboSS := SSeg; TurboSP := SPtr;   { *** !! *** }

  GetIntVec(NEWINT,SaveIntVec);  { Originalwert des zu setzenden Vektors }
  {$IFDEF VER50}
    SetIntVec($3F, SaveInt3F); { V 5.0: Overlay-Vektor zurück }
  {$ENDIF}
  SetIntVec($00, SaveInt00);   { beide Versionen: DIV0-Vektor zurück }

  Writeln('Programm installiert');

  SetIntVec(NEWINT,@IntFrame);   { Installation }

  Keep(0);  { TSR, Exit-Code 0 }
end.
```

Hardware-Interrupts oder: Die Allgemeine Lösung

Die naheliegende Frage, wie man ein bereits eingelagertes Programm mit einem anderen Programm lokalisiert, haben wir im zuvor gezeigten Lösungsansatz recht einfach umgangen: Dort liefert der Aufruf des Interrupts mit einem entsprechenden Funktionscode seine eigene Adresse zurück.

Für TSR-Programme, die keine derartigen Spezialfunktionen definieren können (z.B. deshalb, weil sie zu beliebigen Zeitpunkten und nicht näher definiertem AH-Register aufgerufen werden), müssen wir uns dagegen an den Interrupt-Vektor selbst halten. Woraus auch folgt: Ein Programm wie

```
program Klotz;
uses Dos;
var
  x: array[1..10000] of Integer;
begin
  Keep(0);
end.
```

wird man definitiv nur mit einem Neustart des Systems wieder los, weil es keine Interrupt-Vektoren setzt. (So ganz stimmt diese Behauptung nicht: Das Programm SMAP von Ed Iacobucci findet auch TSR-Programme wie *Klotz* und gibt noch dazu die jeweiligen Namen aus. SMAP ist allerdings als »IBM internal use only« deklariert*) und greift vermutlich auf irgendwelche inoffiziellen Tabellen von DOS zu).

*) Offensichtlich scheint es trotz dieser Deklaration irgendwann einmal in Umlauf geraten zu sein - sonst hätte ich es auch nicht.

Die Version von TRACE86, die in diesem Abschnitt vorgestellt wird, geht von der Überlegung aus, daß die relativen Adressen von Routinen und Daten zweier identischer Programme dieselben sind - unabhängig davon, auf welcher absoluten Adresse sich die beiden Versionen im Speicher befinden.

Ein Beispiel zur Verdeutlichung: Im Hauptspeicher eingelagert sei das Programm A mit den folgenden absoluten Adressen:

```
Codesegment des Hauptprogramms: $2000:0000
            Start von IntFrame: $2000:0185
Datensegment:                   $2580:0000
            Variable X:         $2580:0100
```

Starten wir das Programm A ein zweites Mal - diesmal mit der Absicht, den Wert der Variablen X_r (= $X_{resident}$) herauszufinden. Für diese zweite Programmkopie ergeben sich in unserem Beispiel die Adressen:

```
Codesegment des Hauptprogramms: $3000:0000
            Start von IntFrame: $3000:0185
Datensegment:                   $3580:0000
            Variable X:         $3580:0100
```

Die Startadresse von $IntFrame_r$ ist über den Interrupt-Vektor einfach zu ermitteln und liefert uns so ein Codesegment des eingelagerten Programms. Das Datensegment von A liegt $5800 Bytes oberhalb des Codesegments - und zwar sowohl bei der momentan laufenden Version *als auch beim eingelagerten Programm*: schließlich sind beide Programme identisch und nur auf verschiedenen Segmentadressen gespeichert. Die laufende Version von A erhält die Segmentadresse DS_r also dadurch, daß es seinen eigenen »Abstand« zwischen Code- und Datensegment berechnet und ihn zur Segmentadresse von $IntFrame_r$ dazuaddiert. Mit Pascal-Anweisungen ausgedrückt:

```
GetInterrupt(NEWINT, p);                    { = $2000:0185 }
IntFrameCSr := PWord(p).Seg;                { = $2000 }
Seg_Unterschied := DSeg - Seg(IntFrame);    { = $0580 }
ResDSeg := IntFrameCSr + Seg_Unterschied;   { = $2580 }
```

Die Variable X ist in beiden Versionen des Programms A auf derselben Offset-Adresse innerhalb des Datensegments gespeichert. Die Adresse von X_r ergibt sich deshalb aus dem Datensegment der eingelagerten Version und dem Offset-Anteil von X innerhalb des laufenden Programms, also:

```
p := Ptr(ResDSeg, Ofs(X));                  { = $2580, $0100 }
Writeln('Xr hat den Wert: ', p^);
```

Der langen Vorrede kurzer Sinn: Wenn wir bei der Installation eines Programms die Eckdaten (Originalwert des Interruptvektors, PSP-Segment und ID) in globalen Pascal-Variablen speichern, kommen wir an diese Werte zu einem späteren Zeitpunkt auch dann heran, wenn uns nur der Interruptvektor als Information zur Verfügung steht.

Im Listing von TRACE86R.PAS ist die Routine *IntCheck*, in der die diversen Prüfungen und eventuell das Entfernen einer bereits eingelagerten Version stattfinden, in eine Include-Datei (INTCHECK.INC) verbannt worden*).

```
program TRACE86R;  { 20/12/87 as }
{ Demonstration eines speicherresidenten Programms, das sich
  unabhängig von Funktionscodes wieder aus dem Speicher entfernen läßt. }

{$M 2000,0,0}              { *** !!! *** }

uses Crt, Dos;     { wg. GotoXY, Write bzw. Registers, SetIntVec usw. }

type
  PWord = record    { Typ-Umwandlung für Zeigerwerte }
     Off: Word;        { Offset }
     Seg: Word;        { Segment }
   end;

  ThreeIDs = array[0..2] of Byte; { Typ-Umwandlung für IDs }

const
  IDs: ThreeIDs = (20,12,87);    { ID des Programms }
  OrgIntOff: Word = 0;           { Hier wird der Original- }
  OrgIntSeg: Word = 0;           { Interrupt-Vektor gesetzt }
  Prefix: Word = 0;              { und hier das Programmsegment-Präfix }
  NEWINT = 8;      { Nummer des zu setzenden Interrupts }
```

Die Eckdaten des Programms (ID, Originalvektor und PSP-Segment) sind hier als typisierte Konstanten definiert, wobei weder die Reihenfolge noch die Anfangswerte eine Rolle spielen. Die Typdefinition *ThreeIDs* erleichtert die Prüfung über eine Zeigervariable, die in INTCHECK.INC als *^ThreeIDs* deklariert ist und deshalb nicht für jedes Kennbyte neu gesetzt werden muß.

```
var
  Regs: Registers;
  SSSave, SPSave: Word;     { zum Speichern von SS:SP }
  TurboSS, TurboSP: Word;   { SS:SP von Turbo }
  CallCS,CallIP: Word;      { hier finden sich CS:IP des
                              aufrufenden Programms }
  SaveIntVec: Pointer;      { Original-Vektor }

  CountDown, CountStart: Integer;  { Zeitzähler }
```

Der Deklarationsteil für die globalen Variablen des Programms sieht dem von TRACE86.PAS nicht nur verdächtig ähnlich - bis auf *SaveIntVec* (anstelle von *SaveInt08*) ist er identisch.

*) *IntCheck* wird (in unveränderter Form) noch von mehreren Programmen dieses Kapitels benutzt.

```
procedure IntFrame; external;
{$L INTFRAME}

procedure HexWordPrint(w:Word);
{ Wird von IntProc aufgerufen und gibt den Wert von w in
  hexadezimaler Form (ggf. mit führenden Nullen)
  ab der momentanen Position des Cursors aus. }
var
  x, Chx: Integer;  { Zähler, "Char" }
begin
  for x:= 1 to 4 do  { 4 Stellen }
  begin
    Chx := (w and $F000) shr 12;  { das jeweils oberste Digit }
    w := w shl 4;      { w um eine Hex-Stelle nach links }
    if Chx < 10 then
      Write(chr(Chx + Ord('0')))            { '0'..'9' }
      else Write(chr(Chx - 10 + Ord('A'))); { 'A'..'F' }
  end;
end;

procedure IntProc;
{ Diese Prozedur wird durch den "Clock Tick" aufgerufen }
var
  CursorX, CursorY: Byte;  { für die Originalposition des Cursors }
begin
  { Aufruf der Originalroutine }
    inline($9C/                    { PUSHF }
           $FF/$1E/SaveIntVec);    { CALL FAR [SaveIntVec] }

  Dec(CountDown);              { Zähler }
  if CountDown <= 0 then
  begin
    CursorX := WhereX; CursorY := WhereY;  { Originalposition des Cursors }
    GotoXY(71,1);              { Cursor in die obere rechte Ecke }
    HexWordPrint(CallCS);      { CS-Register des unterbrochenen Programms }
    Write(':');
    HexWordPrint(CallIP);      { IP-Register des unterbrochenen Programms }
    GotoXY(CursorX,CursorY);   { Cursor zurück }
    CountDown := CountStart;   { Zähler wieder auf den Startwert }
  end;
end;
```

Die verdächtige Ähnlichkeit mit TRACE86 gilt auch für *IntProc* und *HexWordPrint* - der einzige Unterschied besteht darin, daß die Originalroutine nicht über *SaveInt08*, sondern über die Variable *SaveIntVec* aufgerufen wird.

```
{$I INTCHECK.INC}        { Setzen von TurboSS, TurboSP, Prüfung auf
                           Kommandozeilen-Parameter OFF usw. }

{ ---------------------------------------------------- }

begin
  if not IntCheck(NEWINT,SPtr-$40) then Halt;

  GetIntVec(NEWINT,SaveIntVec);  { Originalwert des zu setzenden Vektors }

  OrgIntOff := PWord(SaveIntVec).Off;   { Setzen der Eckdaten }
  OrgIntSeg := PWord(SaveIntVec).Seg;
  Prefix := PrefixSeg;

  CountStart := 9;     { zwei Ausgaben pro Sekunde }
  CountDown := 0;      { erste Ausgabe sofort }
  CheckBreak := False; { kein Abbruch via ^C, bitte...}

  {$IFDEF VER50}
    SetIntVec($3F, SaveInt3F);   { V 5.0: Overlay-Vektor zurück }
  {$ENDIF}
  SetIntVec($00, SaveInt00);     { beide Versionen: DIV0-Vektor zurück }
  SetIntVec($1B,SaveInt1B);      { ^Break-Interrupt von CRT zurück }

  Writeln('Programm installiert');

{ *** Achtung: Die Installation muß der letzte Befehl vor KEEP sein -
  ansonsten würde sich das Programm selbst unterbrechen! }

  SetIntVec(NEWINT,@IntFrame);    { Installation der Routine }
  Keep(0);  { TSR, Exit-Code 0 }
end.
```

Zu diesem Programmteil gibt es auch nicht mehr allzuviel zu sagen (auf die Routine *IntCheck* kommen wir gleich noch): Nach der Speicherung der Eckdaten des Programms (Originalvektor in *SaveIntVec*, *OrgIntOff* und *OrgIntSeg*; PSP-Segmentadresse in *Prefix*) wird der Zeitabstand der Ausgaben durch die Zuweisung an *CountStart* festgelegt. Die Erweiterung um einen entsprechenden Kommandozeilen-Parameter dürfte hier ein bißchen mehr Arbeit machen, weil zusätzlich (innerhalb von *IntCheck*) bereits auf einen Parameter (»OFF«) hin geprüft wird.

Schließlich werden die von *System* und *Crt* belegten Interruptvektoren wieder zurückgesetzt und das Programm hängt sich über das Setzen des Vektors $08 (= NEWINT) in den »Clock Tick« ein.

Kommen wir zur Include-Datei INTCHECK.INC und der darin enthaltenen Funktion *IntCheck*. *IntCheck* ist für eine einzelne Routine reichlich lang ausgefallen, was aber seinen Grund hat: Auf diese Weise können wir alle benötigten Variablen

lokal (also auf dem Stack) deklarieren und sparen dementsprechend Speicherplatz für das residente Programm.

IntCheck erwartet zwei Parameter: Die Nummer des zu setzenden bzw. zu überprüfenden Interrupts (*IntNo*) und den Wert für *TurboSP*, der im Hauptprogramm von TRACE86R mit *SPtr-$40* übergeben wird (vgl. Hauptprogramm von TRACE86). Das Ergebnis von *IntCheck* ist nur dann *True*, wenn das Programm tatsächlich installiert werden soll.

```
{ INTCHECK.INC - Include-Datei für TRACE86R.PAS
  und alle DOS-Abfangprogramme   21/12/87 as }

function IntCheck(IntNo, StackTop: Word): Boolean;
const
  OffCmd: string[3] = 'OFF';
var
  Regs: Registers;          { für DOS-Aufrufe }
  CmdStr: string[3];
  Off: Boolean;             { Flag für "Parameter OFF" }
  Installed: Boolean;       { Flag für "bereits installiert" }
  Ch: Char;
  x:Integer;
  EnvSeg: Word;             { Environment-Segmentadresse }
  ResData: Word;            { für die Daten-Segmentadresse des
                              residenten Programms }
  p: ^Word;                 { für diverse Zwecke }
  IDPtr: ^ThreeIDs;         { für die Kennbytes }

begin
  TurboSS := SSeg; TurboSP := StackTop;   { *** UNBEDINGT NOTWENDIG *** }

  { 1. Schritt: Kommandozeilen-Parameter "Off"? }
  if ParamCount >= 1 then
  begin
    Off := True;
    CmdStr:= Copy(ParamStr(1),1,3);
    for x:= 1 to 3 do
      if UpCase(CmdStr[x]) <> OFFCMD[x] then Off:= False;
    if not Off then
    begin
      Writeln('Ungültiger Kommandozeilen-Parameter: ', ParamStr(1));
      IntCheck:= False; Exit;
     end;
   end
  else Off := False;
```

Der »1. Schritt« mit der Prüfung auf den Kommandozeilen-Parameter OFF. Er läuft genauso ab wie im Hauptprogramm von *RemoveableFunctionInterrupt* -

wenn man davon absieht, daß das Programm hier gegebenenfalls nicht mit *Halt* abbricht, sondern lediglich die Funktion *IntCheck* mit dem Ergebnis *False* beendet.

```
{ 2. Schritt: Prüfung, ob das Programm bereits installiert wurde.
  Wenn ja:  a) Kommandozeilen-Parameter "Off" -> hinauswerfen
            b) ansonsten Meldung "bereits installiert" und Abbruch }

Installed := False;
GetIntVec(IntNo,SaveIntVec);
if (PWord(SaveIntVec).Off = Ofs(IntFrame))
  and (SaveIntVec <> nil) then
begin
  { Ermittlung des Datensegments des residenten Programms }
  ResData := PWord(SaveIntVec).Seg      { CSeg des residenten Programms }
             + (DSeg - Seg(IntFrame)); { plus Abstand CSeg | DSeg }

  Installed := True;
  IDPtr := Ptr(ResData,Ofs(IDs));
  for x:= 0 to 2 do
    if IDPtr^[x] <> IDs[x] then Installed := False;

  { Diese vier Daten werden auch dann (unnötigerweise) ermittelt,
    wenn das ID nicht übereinstimmt. }
  p := Ptr(ResData,Ofs(OrgIntOff));   { Original-Interrupt: Offset }
  OrgIntOff := p^;                  { wird in unserem OrgIntOff gespeichert }
  p := Ptr(ResData,Ofs(OrgIntSeg));   { Segment-Adresse }
  OrgIntSeg := p^;
  p := Ptr(ResData,Ofs(Prefix));      { Programmsegment-Präfix }
  Prefix := p^;
  EnvSeg := Word(Ptr(Prefix,$2C)^);   { Environment }
end;
```

Wenn der Offset-Anteil des Interruptvektors und der Offset-Anteil von *IntFrame* übereinstimmen, dann ermittelt *IntCheck* das (vermutliche) Datensegment des residenten Programms und setzt den Zeiger *IDPtr* auf die Adresse, ab der die Kennbytes gespeichert sein sollten. Die Wahrscheinlichkeit einer Verwechslung mit anderen Programmen beim nachfolgenden Vergleich der IDs ist zwar wesentlich höher als bei der Ermittlung über einen Funktionsaufruf (vgl. letzter Abschnitt), liegt aber immer noch jenseits von Gut und Böse, nämlich bei rund 1:60000 * 2^{24}-1.

Die restlichen Eckdaten des residenten Programms werden nach demselben Verfahren ermittelt, d.h. über *ResDSeg* und die jeweiligen Offset-Adressen des laufenden Programms. Damit hier nicht noch ein weiterer **if**-Zweig fällig ist, geschieht diese Ermittlung auch dann, wenn *Installed* bereits den Wert *False* angenommen hat.

```
    if (not Installed) and Off then
    begin
      Writeln('Programm nicht installiert/nicht als erstes in der Kette.');
      IntCheck := False;
      Exit;
    end;
```

Diese Fehlermeldung stellt eine echte Neuerung dar - auf die Probleme, die bei der Verwendung eines Interruptvektors durch mehrere Programme entstehen können, kommen wir im nächsten Abschnitt noch zu sprechen.

```
    if Installed then
    begin
      if not Off then
      begin
        Writeln('Programm ist bereits installiert.');
        IntCheck:= False;
        Exit;
      end
      else begin
        with Regs do
        begin
          AH := $49;        { DOS-Funktion: Free Memory }
          ES := Prefix;
          Intr($21,Regs);  { Freigabe des Programms }
          AH := $49;
          ES := EnvSeg;
          Intr($21,Regs);  { Freigabe des Environments }

          p:= Ptr(OrgIntSeg,OrgIntOff);
          { Und schließlich: Zurücksetzen des Interrupts }
          SetIntVec(IntNo,p);
        end;  { with Regs }
       Writeln('Programm aus dem Speicher entfernt');
       Halt(0);
      end; { else }
    end; { if Installed }

    IntCheck:= True;
  end;
```

Die Entfernung eines installierten Programms läuft nach exakt demselben Schema (und mit denselben Anweisungen) ab wie in *RemoveableFunctionInterrupt*: Zuerst werden PSP-Segment und Programm freigegeben (wobei wir hier *Regs.ES* explizit setzen müssen), dann die dazugehörige *Environment*-Tabelle; schließlich setzt *IntCheck* den Interrupt-Vektor auf den Originalwert zurück.

Vorsicht mit TURBO-eigenen Vektoren!

Wer versucht, auf dem soeben vorgestellten Prinzip eine Behandlungsroutine für CONTROL-BREAK aufzubauen (sprich: den Vektor $1B versetzt) und dabei das Unit *Crt* verwendet, der wird sein blaues Wunder erleben - allerdings erst, wenn er sein Programm wieder aus dem Speicher entfernt: der Vektor $1B bleibt von seiner »Wiederherstellung« reichlich unbeeindruckt und zeigt nach wie vor auf die Adresse, auf die er durch die speicherresidente Version des Programms gesetzt wurde. Der Grund für dieses rätselhafte Verhalten läßt sich am besten anhand einer schrittweisen Ablaufbeschreibung erklären:

1. Start des Programms X_r: Der Initialisierungsteil von *Crt* speichert den Wert des Vektors $1B in der Variablen $SaveIntlB_r$. X_r setzt den Vektor $1B auf eine eigene Behandlungsroutine und endet mit einem Aufruf von KEEP. Wie zu Anfang dieses Kapitels beschrieben, kümmert sich die Prozedur *Keep* nicht um Interruptvektoren - folglich sollte die Sache bis zu diesem Punkt im gewünschten Sinne funktionieren.

2. Zweiter Aufruf des Programms *X*, diesmal mit dem Parameter OFF. Der Initialisierungsteil von *Crt* speichert den momentanen Wert des Interruptvektors $1B in der Variablen *SaveIntlB*.

3. Via *IntCheck* wird der Originalwert des Vektors $1B aus dem Wert der Variablen $SaveIntlB_r$ ermittelt und wieder gesetzt.

4. Diese Version von *X* endet mit einem Aufruf von *Halt*, setzt also die von *System* und *Crt* benutzten Interruptvektoren wieder auf ihre Originalwerte zurück - und damit den Wert des Vektors $1B, der *beim Start der zweiten Version von* X *gesetzt war.*

Erfreulicherweise ist die Beschreibung dieses Problems wesentlich komplexer als seine Lösung, die aus einer zusätzlichen Zeile in INTCHECK.INC besteht:

```
....
   p:= Ptr(OrgIntSeg,OrgIntOff);
   { Und schließlich: Zurücksetzen des Interrupts }
   SetIntVec(IntNo,p);
   SaveIntlB := p;  { Für INT $00: SaveInt00 := p; }
 end;  { with Regs }
Writeln('Programm aus dem Speicher entfernt');
....
```

Nach dieser Zuweisung funkt uns das spätere »Zurückspeichern« der Interruptvektoren (via *Halt*) nicht mehr dazwischen*). Wie im Listing angedeutet, gilt für den Interrupt $00 analoges.

*) Um ganz genau zu sein, wird der vorangehende Aufruf von *SetIntVec* dadurch unnötig und kann gestrichen werden - solche Optimierungen pflegen sich allerdings irgendwann zu rächen.

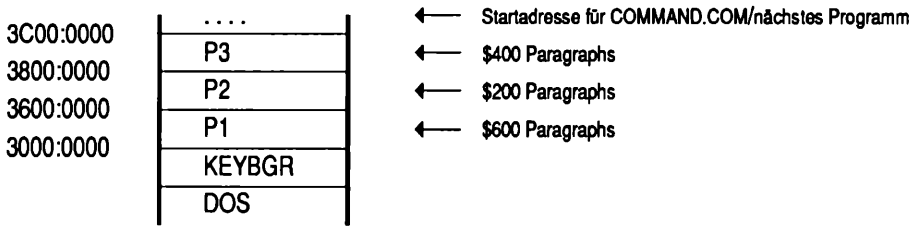
Version 5.0

Hier gilt natürlich dasselbe - nur daß wir es (je nach Modus der Compilierung) mit einer wesentlich höheren Zahl an Interruptvektoren zu tun haben.

Entfernte TSR-Programme, Speicherlöcher und Interruptketten

DOS arbeitet mit einer statischen Speicherverwaltung: Blöcke können belegt, in ihrer Größe verändert und wieder freigegeben werden; ein Umkopieren oder Zusammenschieben mehrerer Blöcke findet nicht statt. Gemessen an anderen Betriebssystemen wie z.B. dem des MacIntosh ist das zwar ein bißchen primitiv, funktioniert aber recht effizient und nach demselben Prinzip wie die Verwaltung des Heaps von Turbo Pascal, die aus einem anderen Grund »dynamisch« heißt.

Zur Demonstration postulieren wir drei speicherresidente Programme (P1, P2 und P3), die nacheinander geladen worden sind:



Wenn wir nun P2 aus dem Speicher entfernen, dann entsteht ein »Loch« von $200 Paragraphs (= $2000 Bytes). DOS prüft von nun an beim Laden jedes Programms, ob diese $2000 Bytes ausreichend sind - wenn ja, wird das Programm zwischen P1 und P2 in den Speicher geladen; ansonsten oberhalb von P3.

Programme, die mit der Version 4.0 oder 5.0 von Turbo Pascal erstellt wurden, benehmen sich ausgesprochen zivilisiert - Aufrufe von Standardfunktionen wie *New* oder *GetMem* bewegen sich nie über den Speicherbereich hinaus, den das Programm bei seinem Start beansprucht hat (vgl. Kapitel 1, Abschnitt »Programmsegment-Präfix«).

Vorsicht! Es gibt eine ganze Reihe von Programmen, die die Obergrenze des RAMs direkt über einen BIOS-Aufruf ermitteln und von der frohen Hoffnung ausgehen, der Rest des Speichers sei ja wohl frei - was in einer Situation wie der eben gezeigten über kurz oder lang zur Zerstörung von P3 führt.

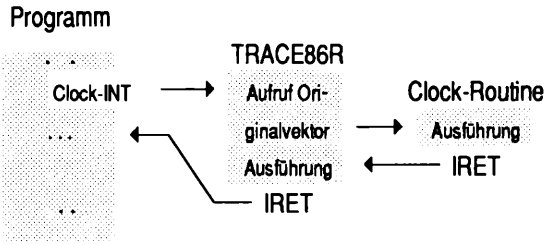
Die Speicherverwaltung von DOS arbeitet auch in einem weiteren Punkt wie der Heap von Turbo Pascal: Benachbarte »Löcher« werden automatisch miteinander verbunden; die Freigabe des obersten Blocks hat ein Herabsetzen des entsprechenden Zeigers und die Löschung darunterliegender Einträge in der Block-Liste zur Folge. Wenn wir im zuvor gezeigten Beispiel nach P2 auch noch P1 wieder aus dem Speicher entfernen, erhalten wir ein durchgehendes Loch von $8000 Bytes, eine anschließende Entfernung von P3 setzt die »nächste freie Adresse« auf das Ende von KEYBGR.

Das *DOS Technical Reference Manual* läßt die Frage, wieviele »Loch«-Einträge die Liste von DOS maximal aufnehmen kann, leider unbeantwortet - einige Dutzend (und damit genug für jeden praktischen Betrieb) sind es aber auf jeden Fall.

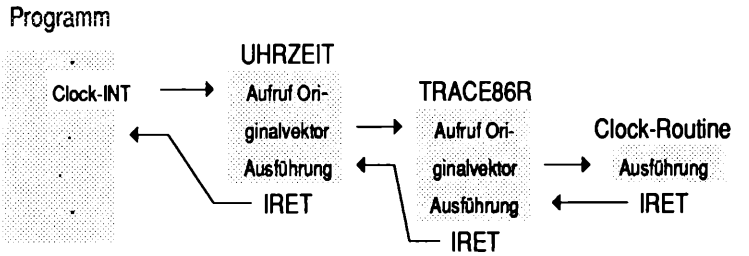
Mehrere Programme über einen Interrupt-Vektor

Mit den bisher beschriebenen Verfahren zum »Einhängen« in einen Interrupt-Ablauf ist es problemlos möglich, mehrere TSR-Programme über ein und denselben Vektor zu steuern - denkbar wäre beispielsweise die Anzeige von CS:IP mit TRACE86R und die Ausgabe der aktuellen Uhrzeit im Sekundentakt. Geben wir diesem zweiten (hypothetischen) Programm den Namen UHRZEIT, nehmen wir an, daß es sich mit demselben Verfahren wie TRACE86R in den »Clock Tick« einhängt, und sehen wir uns den Gesamtablauf an:

- Nach dem Start von TRACE86R:



- Nach dem darauffolgenden Start von UHRZEIT:



Aus diesen beiden Diagrammen lassen sich drei Schlußfolgerungen ableiten:

1. Die Programme werden in der umgekehrten Reihenfolge ihres Starts aufgerufen, weil jedes Programm den Interruptvektor auf sich selbst setzt. Für Programme wie TRACE86R ergibt sich ein unerfreulicher Nebeneffekt: Im zweiten Beispiel bekommen Sie nur noch die (konstante) Adresse innerhalb von UHRZEIT zu sehen, von der aus TRACE86R als »Originalvektor« aufgerufen wird.

Allgemein: Programme, die CS:IP, SS:SP oder das Datensegment eines unterbrochenen Programms auswerten, müssen als *letzte* gestartet werden, damit sie als *erste* in der Interruptkette stehen.

2. Was passiert, wenn Sie nach dem Start von UHRZEIT das Programm TRACE86R wieder aus dem Speicher entfernen wollen? Wenn alles gut geht, gar nichts, weil der Interruptvektor $08 nicht mehr direkt auf TRACE86R, sondern auf das Programm UHRZEIT zeigt. (Hier haben wir den Hauptgrund für die Notwendigkeit von Kennbytes - bei zwei Pascal-Programmen, die beide mit INTFRAME.ASM und INTCHECK.INC arbeiten, ist die Wahrscheinlichkeit doch recht groß, daß die Routine *IntFrame* dieselbe Startadresse innerhalb des Codesegments hat).

Allgemein: TSR-Programme, die denselben Interruptvektor verwenden, müssen in der umgekehrten Reihenfolge ihres Starts wieder aus dem Speicher entfernt werden.

3. Was passiert, wenn Sie nach dem Start von UHRZEIT das Programm TRACE86R ein weiteres Mal in den Speicher einlagern? Da TRACE86R feststellt, daß der Vektor $08 nicht auf eine Kopie seiner selbst zeigt, wird sich das Programm »ordnungsgemäß« installieren. Dieses Spiel ließe sich durch den abwechselnden Start von UHRZEIT und TRACE86R beliebig fortsetzen - bis der Hauptspeicher voll ist.

Allgemein: Solange nicht mit funktionsgesteuerten Interrupts gearbeitet wird, greifen die zuvor besprochenen Schutzmechanismen gegen mehrfache Installationen nur dann, wenn das Programm entweder das einzige ist, das an dem entsprechenden Interruptvektor hängt, oder wenn es als letztes installiert wurde und deshalb zuvorderst in der Kette steht. (Diese beiden Forderungen laufen auf dasselbe heraus: Geprüft werden können nur Programme, auf die der entsprechende Interruptvektor direkt zeigt).

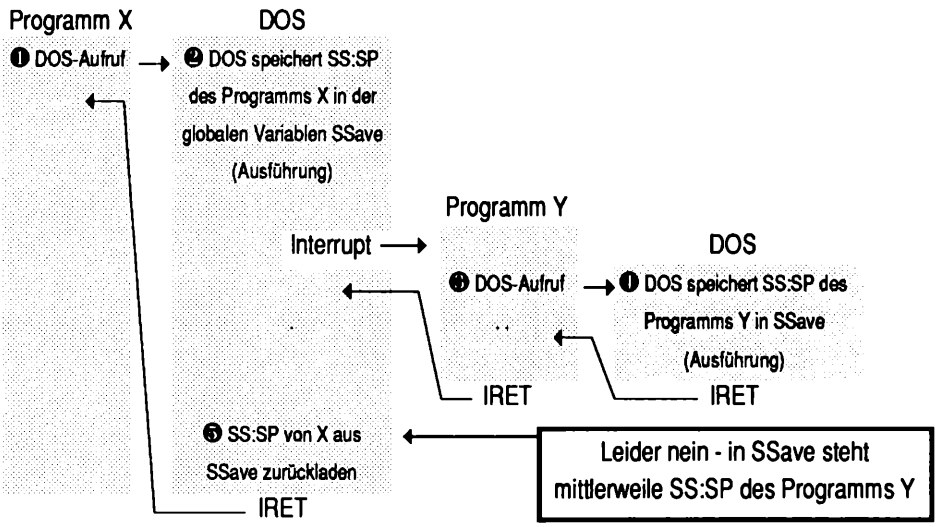
Diese drei Punkte sind deshalb so wichtig, weil wir uns im nächsten Abschnitt mit dem Einbau zusätzlicher Funktionen bei DOS-Aufrufen beschäftigen - und dabei ist es durchaus denkbar, daß ein halbes Dutzend Programme gleichzeitig an ein und demselben Interrupt hängt.

DOS über die Schulter geschaut

Neben dem Einhängen in den »Clock Tick« gibt es eine weitere Möglichkeit, für den zyklischen Aufruf einer Routine zu sorgen: So gut wie jedes Programm führt in unregelmäßigen Abständen Aufrufe von DOS aus. Wenn wir den Interruptvektor $21 verändern und den Aufruf einer Routine auf diese Weise von DOS-Aufrufen abhängig machen, ergeben sich zwei gewaltige Vorteile:

- Die Routine wird nur dann aufgerufen, wenn es wirklich nötig ist - und nicht stumpfsinnig 1 bis 18 Mal in der Sekunde. Wieviel Zeit für die Ausführung benötigt wird, spielt nur noch eine untergeordnete Rolle - es steht nicht zu befürchten, daß das System mit mehreren Routinen zeitlich überladen wird.
- Ein Programm, das »vor« DOS gesetzt wird, kann mit Sicherheit annehmen, daß DOS zum Zeitpunkt seines Aufrufs nicht aktiv ist. Folglich kann das Programm **seinerseits DOS-Aufrufe verwenden.**

Speziell zum zweiten Punkt wäre wohl noch eine Erklärung am Platze: Bis jetzt haben wir die Behauptung unwidersprochen hingenommen, daß TSR-Programme keine DOS-Aufrufe enthalten dürfen. Leider ist sie uneingeschränkt zutreffend: DOS arbeitet mit einem ähnlichen Verfahren zum Setzen eines eigenen Stacks wie unser Programm *IntFrame*. Bei der Unterbrechung eines laufenden DOS-Aufrufs durch eine Routine, die ihrerseits DOS-Aufrufe durchführt, besteht der Weg ins Nirwana aus fünf Schritten:



Nachdem das Programm X nun mit dem Stack von Y weiterarbeitet, führt spätestens der nächste RET-Befehl innerhalb von X in die bereits erwähnten Gefilde.

Noch unerfreulicher: Die meisten BIOS-Routinen werden ebenfalls über DOS aufgerufen. Ein Programm, das beispielsweise Ausgaben an einem Drucker (BIOS-Interrupt $17) in einer Datei speichern will, funktioniert deshalb nur dann, wenn das aufrufende Programm diesen Interrupt direkt (und nicht über einen DOS-Aufruf) benutzt - worauf man sich lieber nicht verlassen sollte.

Eine Anmerkung für Spezialisten: Es gibt zwei interne DOS-Interrupts, über die man DOS-Funktionen ohne Umspeichern von SS:SP aufrufen kann. Diese Interrupts werden beispielsweise von SideKick Plus verwendet - einem Programm, das insgesamt die Vektoren $08, $09, $0B, $0D, $10, $13, $15, $16, $1A, $1C, $21, $25, $26, $28 und $33 belegt. Wir bleiben lieber bei etwas kleineren Brötchen.

Zyklischer Aufruf über DOS-Funktionen

Das Programm DOSHOOK.ASM demonstriert die erste Möglichkeit: die Aktivierung einer Pascal-Routine namens *PasProc* bei jedem Aufruf einer bestimmten DOS-Funktion. Eine Übergabe von Registerwerten findet dabei *nicht* statt - schließlich soll *PasProc* auch keine DOS-Funktion ersetzen, sondern nur zu einem »sicheren Zeitpunkt« aufgerufen werden (d.h. außerhalb aller Aktivitäten von DOS).

Die Routine *IntFrame* innerhalb von DOSHOOK.ASM wird (wie gehabt) vom Pascal-Programm als Ziel des Interrupts gesetzt. Sie führt sieben Schritte aus:

1. Prüfung, ob die DOS-Funktion *FuncNo* aufgerufen werden soll. (*FuncNo* ist eine Pascal-Variable - auf diese Weise läßt sich ein und dieselbe Version von DOSHOOK für mehrere Programme verwenden). Wenn das Ergebnis negativ ausfällt, geht es sofort mit dem tatsächlichen DOS-Aufruf weiter.
2. Zwischenspeichern aller Register, Setzen des Datensegments und des Stacks von Turbo.
3. Setzen des originalen DOS-Interrupts. DOS-Aufrufe durch die Prozedur *PasProc* führen also nicht wieder zu *IntFrame*, sondern direkt zu DOS.
4. Aufruf von *PasProc* ohne Parameter und/oder Registerwerte.
5. Der Interruptvektor $21 wird wieder auf seinen vorherigen Wert gesetzt (d.h. auf *IntFrame*, wenn DOSHOOK das einzige Programm außer DOS selbst ist, das an diesem Vektor hängt).
6. Wiederherstellen der Register, des Datensegments und des Stacks.
7. Sprung zu DOS, d.h. Ausführung der eigentlich aufgerufenen Funktion.

Das Listing:

```
; DOSHOOK.ASM { 22/12/87 as }
; Voranstellen einer Pascal-Routine vor eine DOS-Funktion

DosVec  equ 0084h                ; Adresse des DOS-Interruptvektors
DATA    Segment Word PUBLIC
SSSave  dw 0                     ; zum Speichern von SS:SP
SPSave  dw 0
        EXTRN TurboSS:Word,TurboSP:Word   ; SS:SP von Turbo
        EXTRN  FuncNo: Byte      ; Nummer der DOS-Funktion
DATA    ends
```

Hier sind nur *TurboSS*, *TurboSP* und die neue Variable *FuncNo* als EXTRN deklariert - weil alle Register auf dem (Turbo-)Stack gespeichert werden, kommt DOSHOOK ohne weitere Pascal-Variablen aus. Da wir *PasProc* sowieso keine weiteren Informationen liefern, sind *SSSave* und *SPSave* innerhalb von DOSHOOK.ASM

deklariert; das Flag-Register und CS:IP des aufrufenden Programms bleiben auf dem Stack, d.h. *CallIP* und *CallCS* fehlen völlig.

```
CODE    Segment Byte PUBLIC
        PUBLIC IntFrame
        PUBLIC DosOrg           ; originaler DOS-INT
        EXTRN  PasProc:near     ; die aufzurufende Pascal-Prozedur
        ASSUME cs:CODE, ds:DATA

IntData dw DATA                 ; Turbo-Datensegment
DosOrg  dw 0,0                  ; Originalvektor zu DOS, von Turbo gesetzt
```

DosOrg ist als PUBLIC deklariert und deshalb für das Pascal-Programm verfügbar - an dieser Stelle muß bei der Initialisierung der Originalvektor zu DOS eingetragen werden.

```
IntFrame proc FAR
        push ds
        mov ds,IntData          ; DS aufs Turbo-Datensegment
        cmp ah,FUNCNO           ; sollen wir aktiv werden?
        jz CallTurbo
        pop ds
        push [DosOrg+2]
        push [DosOrg]
        ret                     ; nein.. ab zu DOS
```

Wenn die DOS-Funktionsnummer im AH-Register nicht mit dem Wert der Pascal-Variablen *FuncNo* übereinstimmt, geht es gleich mit einem Sprung zur tatsächlichen DOS-Funktion weiter: Die Flags befinden sich noch unverändert auf dem Stack, als »Rücksprungadresse« wird der Originalvektor dazugeladen. Da *IntFrame* als *far* deklariert ist, codiert der Assembler die ret-Anweisung als RET FAR.

Der Umweg über den Stack ist eigentlich unnötig, theoretisch sollte der Befehl `jmp far [DosOrg]` denselben Zweck erfüllen. Daß dem nicht so ist, führt zu einem **Vorsicht!** In einer Anweisung wie `jmp cs:[xxxx]` wird `xxxx` von MASM korrekt als zu relozierendes Symbol gespeichert, wovon man sich durch einen Blick auf das Listing überzeugen kann. *Der Linker von Turbo Pascal ignoriert derartige Symbole ohne Fehlermeldung* (siehe auch Anhang D).

```
CallTurbo: mov [SSSave],ss      ; SS:SP speichern
           mov [SPSave],sp
           cli                  ; (eigentlich unnötig, s. Anhang A)
           mov ss,[TurboSS]     ; Stack von Turbo setzen
           mov sp,[TurboSP]
           sti
           push ax
           push bx
           push cx
```

```
push dx                ; alle Register auf den Stack
push bp
push si
push di
push es
```

Der zweite Schritt: Nachdem das Datensegment des Pascal-Programms bereits bei der Prüfung der DOS-Funktionsnummer gesetzt wurde, folgt hier das Umsetzen des Stacks und die Speicherung aller Register.

```
xor ax,ax
mov es,ax              ; ES auf Segment 0000
push es:[DosVec]       ; momentanen INT 21-Vektor
push es:[DosVec+2]     ; auf den Stack
mov ax,[DosOrg]
mov es:[DosVec],ax     ; DOS-Interruptvektor auf DOS
mov ax,[DosOrg+2]
mov es:[DosVec+2],ax

call PASPROC           ; Aufruf der Pascal-Routine

xor ax,ax
mov es,ax             ; ES wieder auf Segment 0000
pop es:[DosVec+2]     ; INT 21-Vektor wieder zurück
pop es:[DosVec]
```

Die Schritte 3, 4 und 5: Zwischenspeicherung des Originalvektors, Setzen dieses Vektors auf DOS und Aufruf von *PasProc*. Danach wird der Interruptvektor $21 wieder auf *IntFrame* gesetzt.

Achtung: Wir speichern hier den Interruptvektor $21 auf dem Stack und setzen ihn danach vom Stack aus wieder - wieso schreiben wir nicht nach dem Aufruf von *PasProc* einfach wieder die (bekannte) Adresse von *IntFrame* hinein? *Weil der Vektor gar nicht auf* IntFrame *zeigen muß.* Es könnte genausogut sein, daß dieses Programm von einem anderen aus als »DOS-Originalvektor« aufgerufen wird (vgl. Abschnitt »Mehrere Programme über einen Vektor« in diesem Kapitel).

```
pop es
pop di
pop si
pop bp
pop dx
pop cx
pop bx
pop ax
```

```
            cli                 ; <- dieses CLI ist auf jeden Fall notwendig
            mov ss,[SSSave]     ; SS:SP zurück
            mov sp,[SPSave]
            sti
            pop ds
```

Der sechste Schritt: Zurückladen der Registerwerte, des originalen Stacks und des Datensegments.

```
            push [DosOrg+2]
            push [DosOrg]
            ret                 ; Sprung zu DOS
IntFrame    endp
CODE        ends
            end
```

Der siebte und letzte Schritt: Auf dem (Original-)Stack befindet sich nach wie vor das Flag-Register des Programms, das den Interrupt $21 ausgelöst hat. Der Originalvektor zu DOS wird als »Rücksprungadresse« dazugeladen, mit RET (FAR) geht es weiter zu DOS.

DosOrg ist Teil des CODE-Segments, der Assembler spricht diese Adresse deshalb über das CS-Register an. *Diese* Art der Adressierung verdaut der Linker von Turbo Pascal (im Gegensatz zu einem `jmp far cs:[DosOrg]`) ohne Probleme - Gott sei Dank, denn sonst würde der eigentliche DOS-Aufruf in eine arge Stack-Trickserei ausarten.

Ein Demonstrationsprogramm zu DOSHOOK

Das erste Demonstrationsprogramm zu DOSHOOK.ASM trägt den bezeichnenden Namen HOOKDEMO.PAS. Seine Prozedur *PasProc* wird bei jedem Aufruf der DOS-Funktion $0A (»Buffered Keyboard Input«) aktiviert - also beispielsweise dann, wenn COMMAND.COM eine Eingabe erwartet. *PasProc* ist wieder einmal recht rudimentär ausgefallen und gibt das Zeichen `chr(1)` oben links im Bildschirm aus*). Und wo sind die DOS-Aufrufe innerhalb von *PasProc*? Um ehrlich zu sein nirgends - da das Programm ausschließlich zur Überprüfung von DOSHOOK.ASM dienen soll, wurde darauf verzichtet, irgendetwas außer einem Lebenszeichen in *PasProc* hineinzukonstruieren.

HOOKDEMO verwendet die Include-Datei INTCHECK.INC in unveränderter Form (siehe Seite 256) und läßt sich durch einen Aufruf mit dem Kommandozeilen-Parameter OFF wieder aus dem Speicher entfernen.

*) Ich verspreche es: zum letzten Mal.

```
program HookDemo;  { HOOKDEMO.PAS, 22/12/87 }

{$M 2000,0,0}              { HeapMax = 0 - UNBEDINGT NOTWENDIG }

uses Dos;    { wg. Registers, SetIntVec usw. }
type
  PWord = record   { Typ-Umwandlung für Zeigerwerte }
     Off: Word;       { Offset }
     Seg: Word;       { Segment }
   end;

  ThreeIDs = array[0..2] of Byte; { Typ-Umwandlung für IDs }

const
  IDs: ThreeIDs = (22,12,87);   { ID des Programms }
  OrgIntOff: Word = 0;          { Hier wird der Original- }
  OrgIntSeg: Word = 0;          { Interrupt-Vektor gesetzt }
  Prefix: Word = 0;             { und hier das Programmsegment-Präfix }

  FUNCNO:Byte = $0A;  { Bei dieser DOS-Funktion wird DOSPROC eingeschoben }
  DOSINT = $21;  { Dieser Interrupt wird ersetzt }
```

Die Deklaration der Konstanten und Datentypen enthält nur eine Neuigkeit: Über *FUNCNO* wird die Nummer der DOS-Funktion festgelegt, bei deren Aufruf *PasProc* aktiviert werden soll.

```
var
  TurboSS, TurboSP: Word;  { SS:SP von Turbo }
  SaveIntVec: Pointer;     { zum Speichern des Originalvektors }
  p: ^Pointer;
```

Der Variablen-Deklarationsteil ist erheblich geschrumpft - hauptsächlich deshalb, weil wir *PasProc* keine Registerwerte des unterbrochenen Programms übermitteln.

```
procedure IntFrame; external;
procedure DosOrg; external;     { muß auf den DOS-Originalvektor gesetzt
                                  werden }
{$L DOSHOOK}

procedure PasProc;   { diese Prozedur wird eingeschoben }
begin
   Mem[$B800:0] := 1;   { $B000:0 für Monochrom, Hercules }
end;

{$I INTCHECK.INC}   { Prüfung auf Kommandozeilen-Parameter OFF, Setzen
                      der diversen globalen Variablen usw. }
```

```
{ ------------------------------------------------------- }
begin
  if not IntCheck(DOSINT,SPtr) then Halt;

  GetIntVec(DOSINT,SaveIntVec);  { Originalwert des zu setzenden Vektors }

  OrgIntOff := PWord(SaveIntVec).Off;
  OrgIntSeg := PWord(SaveIntVec).Seg;
  Prefix := PrefixSeg;

  {$IFDEF VER50}
    SetIntVec($3F, SaveInt3F); { V 5.0: Overlay-Vektor zurück }
  {$ENDIF}
  SetIntVec($00, SaveInt00);   { beide Versionen: DIV0-Vektor zurück }

  Writeln('Programm installiert');

  p := @DosOrg; p^:=  SaveIntVec;  { Eintrag des DOS-Originalvektors }

{ *** Achtung: Die Installation muß der letzte Befehl vor KEEP sein -
  ansonsten würde sich das Programm selbst unterbrechen! }

  SetIntVec(DOSINT,@IntFrame);    { Installation der Routine }
  Keep(0);  { TSR, Exit-Code 0 }
end.
```

Nach der Überprüfung auf den Kommandozeilen-Parameter OFF, bereits geladene Versionen des Programms usw. durch *IntCheck* führt das Hauptprogramm drei Schritte aus:

1. Speicherung der Eckwerte (Originalvektor und PSP-Segment) für eine spätere Entfernung aus dem Hauptspeicher.
2. Eintrag des DOS-Originalvektors in die »Prozedur« *DosOrg*, die sich im CODE-Segment von DOSHOOK.ASM befindet. Über diesen Wert findet später der tatsächliche DOS-Aufruf statt.
3. Umsetzen des Interruptvektors $21 auf die Prozedur *IntFrame*. Diese Anweisung muß nicht unbedingt die letzte vor KEEP sein - ein nachfolgender Aufruf der DOS-Funktion, über die *PasProc* aktiviert wird (für unser Beispiel: *Readln*) wäre allerdings tödlich (vgl. TRACE86R).

   Ein Programm, das über den Aufruf der DOS-Funktion KEEP (FUNCNO = $31) aktiviert wird, läßt sich dennoch konstruieren: Ähnlich wie bei TRACE86R müssen dazu die obersten rund $40 Bytes des Stacks reserviert bleiben.

Probieren Sie das Programm einmal aus: Während langer Directory-Listings oder der Ausgabe einer Textdatei mit `type` verschwindet die Spielkarte vom Bildschirm - und erscheint wieder, sobald COMMAND.COM die nächste Eingabe anfordert.

Im Gegensatz zu *Smiley2* und seinen über den Interrupt $08 gesteuerten Nachfolgern macht sich HOOKDEMO innerhalb von Programmen wie der integrierten Entwicklungsumgebung überhaupt nicht bemerkbar, weil TURBO.EXE die DOS-Funktion $0A an keiner Stelle verwendet.

Nicht nur COMMAND.COM, sondern auch DEBUG, SYMDEB und eine Vielzahl anderer Programme verlassen sich dagegen auf die DOS-Funktion $0A - und damit sind wir beim Thema des nächsten Abschnitts.

Abfangen von BIOS-Funktionen über DOSHOOK

Die zuvor aufgestellte Behauptung, daß das Abfangen von BIOS-Aufrufen nur dann möglich ist, wenn dabei keine DOS-Aufrufe im Spiel sind, soll hier nicht zurückgenommen werden - für bestimmte Anwendungen gibt es aber einen einfachen Ausweg.

Ein Problem, das sich unsereinem immer wieder stellt, ist das Abfangen von Ausgaben an den Drucker und ihre Speicherung in einer Datei - egal, ob es sich dabei um Assembler-Listings handelt oder die Dokumentation von Ausgaben eines Programms. Einige der Vorüberlegungen bei der Suche nach einer universellen Lösung sind:

- DEBUG, SYMDEB usw. erlauben DOS das Abfangen des Tastenbefehls CONTROL-P und unterstützen damit die simultane Ausgabe auf einen Drucker. Diese Ausgabe läßt sich über eine erneute Eingabe von CONTROL-P sogar selektiv steuern, eine gezielte Dokumentation von Programmteilen ist also möglich. Mit der Nachbearbeitung dieser Ausgaben sieht es allerdings schlecht aus.

- Dieselben Programme unterstützen die Umleitung der Ausgabe in eine Datei, d.h. Aufrufe der Form *Programmname > Zieldatei*. Von gezielter Dokumentation kann hier allerdings nicht die Rede sein, weil *sämtliche* Ausgaben in der Zieldatei landen.

- Programme wie dBASE III oder Turbo Basic, die Befehle zur direkten Ansteuerung eines Druckers kennen, lassen sich verständlicherweise weder mit CONTROL-P noch mit *Programmname > Zieldatei* beeindrucken. Eine zusätzliche Komplikation ergibt sich dadurch, daß der Drucker in einigen Fällen über Aufrufe von DOS, in anderen Fällen über BIOS-Interrupts angesteuert wird.

Soviel zur Begründung, warum ich mir ausgerechnet ein Problem ausgesucht habe, das »eigentlich keins ist«. BIOS-Umleitungen über DOSHOOK haben zwei Prinzipien zur Grundlage:

1. Der BIOS-Interruptvektor ist auf eine als **interrupt** deklarierte Pascal-Prozedur gesetzt, die die gewünschte Funktion durchführt, aber *ohne DOS-Aufrufe* arbeitet. Ausgaben oder ähnliches werden in einem Puffer zwischengespeichert.
2. Die Prozedur *PasProc* wird bei DOS-Aufrufen aktiviert und prüft von der **interrupt**-Prozedur gesetzte Flags. Innerhalb von *PasProc* werden DOS-Funktionen ausgeführt wie z.B. das Schreiben eines Pufferinhalts auf die Diskette.

Das Programm *PrinterToFile* arbeitet nach diesen Prinzipien: Der Interruptvektor für die BIOS-Funktion zur Ansteuerung des Druckers ist auf die (als **interrupt** deklarierte) Pascal-Prozedur *SaveChar* gesetzt, die nichts anderes tut, als das auszugebende Zeichen in einem Puffer zu speichern und den Ergebniscode der BIOS-Funktion zu simulieren. (Umfangreichere Operationen darf diese Routine wegen der Stackprobleme sowieso nicht durchführen).

PasProc wird bei dem Aufruf der DOS-Funktion »Buffered Keyboard Input« aktiviert und prüft, ob *SaveChar* zwischenzeitlich Zeichen im Puffer abgelegt hat - wenn ja, wird der Pufferinhalt in die Datei PRTFILE auf die Diskette geschrieben.

```
program PrinterToFile;  { PRTFILE.PAS - 12/87, 12/88 as }
{ Fängt Ausgaben zum Drucker ab und speichert sie in einer Datei
  namens PRTFILE.

  Über den Aufruf PRTFILE OFF läßt sich das Programm wieder aus dem
  Speicher entfernen.  }

{$I-,S-,R-}        { keine Prüfung von I/O, Stack und Bereichen }
{$M 2000,0,0}              { *** !!! *** }

uses Crt, Dos;    { wg. GotoXY bzw. Registers, SetIntVec usw. }
type
  PWord = record   { Typ-Umwandlung für Zeigerwerte }
     Off: Word;       { Offset }
     Seg: Word;       { Segment }
   end;
  ThreeIDs = array[0..2] of Byte; { Typ-Umwandlung für IDs }

const
  IDs: ThreeIDs = (25,12,87);   { ID des Programms }
  OrgIntOff: Word = 0;          { Hier wird der Original- }
  OrgIntSeg: Word = 0;          { Interrupt-Vektor gesetzt }
  Prefix: Word = 0;             { und hier das Programmsegment-Präfix }

  BUFSIZE = 32767;     { Größe des Zeichenpuffers }
  FUNCNO:Byte = $0A;  { Bei dieser DOS-Funktion wird DOSPROC eingeschoben }
  DOSINT = $21;  { Dieser Interrupt wird ersetzt }
  PRTINT = $17;  { der auch }
```

Dieser Teil enthält gegenüber den vorangegangenen Beispielprogrammen nur zwei Neuigkeiten: Zum einen wird die Prüfung der Ein-/Ausgaben hier explizit abgeschaltet, zum anderen definieren wir *zwei* Konstanten für Interruptnummern (DOSINT und PRTINT). Die Konstante FUNCNO legt fest, daß *PasProc* bei Aufrufen der DOS-Funktion »Buffered Keyboard Input« aktiv wird; zur Speicherung von Zeichen zwischen zwei Aufrufen von *PasProc* stellt das Programm 32 KByte zur Verfügung.

```
var
  TurboSS, TurboSP: Word;  { SS:SP von Turbo }
  SaveIntVec: Pointer;     { zum Speichern des Originalvektors von DOS }
  SaveInt2Vec: Pointer;    { zum Speichern des Drucker-Originalvektors }
  p: ^Pointer;

  OurInt24: Pointer;       { für "Critical Errors" }
  Buffer: array[0..BUFSIZE] of Char;  { Zwischenspeicherung der Ausgaben }
  BufPos: Integer;         { Index in Buffer }
  OutFile: Text;           { Ausgabedatei }
```

TurboSS und *TurboSP* müssen wie mittlerweile üblich auf das obere Ende des Turbo-Stacks gesetzt werden, *SaveIntVec* und die neue Variable *SaveInt2Vec* nehmen die Originalwerte der veränderten Interruptvektoren auf.

Die Deklaration von *Buffer*, *BufPos* und *OutFile* ist (im Gegensatz zu den restlichen Variablen) programmspezifisch - genauso wie der Zeiger *OurInt24*, auf den wir auf der nächsten Seite noch zurückkommen.

```
procedure IntFrame; external;
procedure DosOrg; external;     { muß auf den DOS-Originalvektor gesetzt
                                  werden }
{$L DOSHOOK}
```

Der Aufruf von *PasProc* erfolgt wie bei HOOKDEMO über DOSHOOK.ASM (dieses Programm ist im vorhergehenden Abschnitt aufgelistet). Die »Prozedur« *DosOrg* befindet sich im CODE-Segment von DOSHOOK.ASM und besteht nur aus vier reservierten Speicherzellen, in die der DOS-Originalvektor durch das Pascal-Hauptprogramm eingesetzt wird.

```
procedure PasProc;   { diese Prozedur wird eingeschoben }
var x: Integer;
    CursorX, CursorY: Byte;  { zum Speichern der Cursorposition }
begin
  if BufPos <> 0 then
  begin
    CursorX:= WhereX; CursorY := WhereY;  { Originalposition }
    GotoXY(1,1); Write('PRTFILE  ');      { ein Lebenszeichen }
    GotoXY(1,2); Write('         ');
    GotoXY(CursorX, CursorY);             { Cursor zurück }
```

```
    GetIntVec($24, SaveInt24);     { Speichern des mom. INT $24 }
    SetIntVec($24, OurInt24);      { und Setzen des eigenen Handlers }

    Append(OutFile);
    if IOResult <> 0 then Rewrite(OutFile);

    if IOResult = 0 then     { ansonsten bleibt der Puffer, wo er ist }
    begin
      for x:= 0 to BufPos - 1 do      { zeichenweises Schreiben }
        Write(OutFile, Buffer[x]);    { des Pufferinhalts }
      Close(OutFile);
      BufPos:= 0;
    end; { if IOResult }
    SetIntVec($24, SaveInt24);     { INT $24 zurück }
  end;  { if BufPos }
end;
```

PasProc wird bei jedem Aufruf der DOS-Funktion $0A aktiviert und prüft anhand von *BufPos*, ob *SaveChar* zwischenzeitlich Zeichen in *Buffer* gespeichert hat. Nur wenn das der Fall ist, wird a) der String PRTFILE oben links im Bildschirm ausgegeben und b) der Pufferinhalt über die Textdatei-Variable *OutFile* auf die Diskette geschrieben. (*OutFile* wird über das Hauptprogramm der Name 'PRTFILE' zugeordnet).

Die Kombination aus *Append* und *Rewrite* ist recht unempfindlich gegenüber plötzlich verschwundenen Dateien, gewechselten Directories und anderen Widrigkeiten - auch ein Löschen von PRTFILE mit `del`, während das Programm aktiv ist, hat schlicht die Erzeugung einer neuen Datei zur Folge.

Was die Manipulationen des Interruptvektors $24 betrifft: Da *System* eine eigene Routine für »Critical Errors« definiert, bleibt einem die berüchtigte Meldung `<A>bort <R>etry <I>gnore` normalerweise auch bei offenen Laufwerksklappen usw. erspart - es sei denn, das Turbo Pascal-Programm ist speicherresident: Wie zu Anfang dieses Kapitels kurz angesprochen, setzt DOS den Vektor $24 auch bei einem Programmende via KEEP auf seinen Originalwert (sprich: die zuvor erwähnte Dreibuchstaben-»Fehlerkorrektur«) zurück.

Wenn wir nun noch die Tatsache dazunehmen, daß DOS diese Meldung (nach einem CONTROL-P) auch auf einen Drucker ausgibt, dann lassen sich wunderliche Endlosschleifen konstruieren: Nach dem Wechsel auf ein nicht betriebsbereites Laufwerk gibt DOS eine erste Meldung aus, die PRTFILE danach *auf diesem Laufwerk* speichern will - was natürlich zu einer weiteren Fehlermeldung führt, die PRTFILE wiederum als »Drucker-Echo« mitbekommt, erneut schreiben will und dabei mit einem DOS-Fehler endet, der seinerseits*) ...

Genug der üblen Scherze. In *System* steht uns eine eigene Behandlungsroutine für »Critical Errors« zur Verfügung, deren Adresse durch das Hauptprogramm von

*) Erstaunlicherweise kommt man mit sechs oder sieben Eingaben von <I>gnore (gefolgt von einem schleunigsten Wechsel des Laufwerks) aus diesem Karussell relativ folgenlos wieder heraus.

PRTFILE in der Variablen *OurInt24* gespeichert und jeweils für die Dauer der Ausführung von *PasProc* gesetzt wird. Sollte sich ein eventueller Fehler auch durch ein *Rewrite* nicht beseiten lassen, bleiben Puffer und Pufferindex einfach auf dem alten Stand - PRTFILE probiert dann beim nächsten Aufruf von neuem, die gespeicherten Zeichen zu schreiben.

```
procedure SaveChar(Flags, CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP: Word); interrupt;
{ Wird über den Drucker-Interrupt aufgerufen }
begin
  if AX and $FF00 = 0 then  { Zeichenausgabe }
  begin
    if BufPos < BufSize then
    begin
      Buffer[BufPos] := Chr(AX);  { AH ist 0 }
      Inc(BufPos);
     end;
  end;
  AX := (AX and $00FF) or ($9000);  { nicht "Busy", "ist selektiert" }
end;
```

Die als **interrupt** deklarierte Prozedur *SaveChar* ersetzt den BIOS-Interrupt $17 und wird bei jeder Ausgabe an den Drucker aufgerufen. Dieser Interrupt kennt drei Subfunktionen, die sämtlich den Druckerstatus im AH-Register als Ergebnis haben:

AH = 0 Ausgabe des Zeichens in AL - solange *Buffer* noch nicht randvoll ist, wird das Zeichen von *SaveChar* dort gespeichert.

AH = 1 Initialisierung der Druckerschnittstelle - hat bis auf das Zurückliefern des Statuscodes keine weiteren Aktionen von *SaveChar* zur Folge.

AH = 2 Zurückliefern des Druckerstatus - mehr tut *SaveChar* in diesem Fall auch nicht.

Unabhängig vom Funktionscode wird nur der Inhalt von AX verändert, alle anderen Register bleiben erhalten.

```
{ ---------------------------------------------------------- }

{$I INT2CHK.INC}          { Prüfung auf OFF usw. Ist gegenüber INTCHECK
                            um einen zusätzlichen Parameter erweitert }
```

Weil wir hier gleich mit zwei Interruptvektoren auf einmal arbeiten, kommen wir leider um eine neue Version der Funktion *IntCheck* nicht herum. Erfreulicherweise beschränken sich die Änderungen gegenüber der Datei INTCHECK.INC auf sieben Zeilen, sie sind auf den folgenden Seiten besprochen.

```
begin
  if not IntCheck(DOSINT,PRTINT,SPtr) then Halt;  { ** 3 Parameter }

  GetIntVec(DOSINT,SaveIntVec);  { Originalwert des DOS-Vektors }
  OrgIntOff := PWord(SaveIntVec).Off;
  OrgIntSeg := PWord(SaveIntVec).Seg;

  GetIntVec(PRTINT,SaveInt2Vec);  { Originalwert des Drucker-Vektors }

  Prefix := PrefixSeg;

  CheckBreak := False;     { kein Abbruch via ^C, bitte... }

  {$IFDEF VER50}
    SetIntVec($3F, SaveInt3F); { V 5.0: Overlay-Vektor zurück }
  {$ENDIF}
  SetIntVec($00, SaveInt00);   { beide Versionen: DIV0-Vektor zurück }
  SetIntVec($1B,SaveInt1B);       { ^Break-Vektor von CRT zurück }

  GetIntVec($24, OurInt24);    { SYSTEM-Behandlungsroutine für
                                 "Critical Errors" }

  Writeln('PRTFILE (c) as 87/88: Programm installiert');

  Assign(OutFile,'PRTFILE'); BufPos := 0;

{ *** Achtung: Die Installation muß der letzte Befehl vor KEEP sein -
  ansonsten würde sich das Programm selbst unterbrechen! }

  p := @DosOrg; p^:=  SaveIntVec; { für den Sprung zu DOS in DOSHOOK }
  SetIntVec(PRTINT,@Savechar);    { Installation von SaveChar }
  SetIntVec(DOSINT,@IntFrame);    { Installation der Routine }
  Keep(0);  { TSR, Exit-Code 0 }
end.
```

Das Hauptprogramm von PRTFILE muß nicht nur den Originalwert des DOS-Vektors und sein PSP-Segment, sondern auch den Vektor zum BIOS-Interrupt $17 speichern und danach neu setzen.

Die Adresse der Behandlungsroutine für »Critical Errors« in *System* ist nirgendwo öffentlich deklariert. Wir wissen aber, daß *System* den Vektor $24 auf diese Routine gesetzt hat - folglich stellt der entsprechende Aufruf von *GetIntVec* nicht nur einen sicheren, sondern auch versionsunabhängigen Weg zu ihrer Ermittlung dar (so merkwürdig die Sache auf den ersten Blick auch aussehen mag).

Ansonsten findet sich hier bis auf die Zuordnung des Namens PRTFILE zur Dateivariablen *OutFile* und die Initialisierung von *BufPos* auf 0 (= »Puffer leer«) nichts Neues.

Der Inhalt der Datei INT2CHK, d.h. die Funktion *IntCheck* ist hier noch einmal komplett mit allen Erläuterungen wiedergegeben, weil ich nicht davon ausgehe, daß jeder Leser dieses Kapitel von der ersten bis zur letzten Zeile durchgearbeitet hat. Die Änderungen gegenüber der Datei INTCHECK.INC sind fett gesetzt.

Die Funktion *IntCheck* erwartet in dieser Version drei Parameter: die Nummern der zu setzenden Interrupts und die Größe des Stacks. Das Funktionsergebnis ist nur dann *True*, wenn das Hauptprogramm fortgesetzt werden soll, d.h. weder der Kommandozeilen-Parameter OFF angegeben wurde noch bereits eine Kopie dieses Programms im Speicher installiert ist. *IntCheck* besteht im wesentlichen aus drei Schritten:

- Prüfung auf den Kommandozeilen-Parameter OFF;
- Prüfung, ob bereits eine Kopie des laufenden Programms installiert ist.
- Wenn bereits eine Kopie installiert ist: Wenn der Parameter OFF angegeben wurde, Entfernung dieser Kopie aus dem Speicher; ansonsten Ausgabe der Meldung »Programm bereits installiert«.

(Vernünftigerweise hätte man dieser Version von *IntCheck* auch einen anderen Namen geben sollen - wenn Sie sich die Mühe machen wollen, müssen Sie zusätzlich alle Zuweisungen an das Funktionsergebnis entsprechend ändern).

```
{ INT2CHK.INC - 25/12/87 as }
{ Include-Datei für PRTFILE.PAS und TRACE287.PAS }

function IntCheck(IntNo, Int2No, StackTop: Word): Boolean;
const
  OffCmd: string[3] = 'OFF';
var
  Regs: Registers;          { für DOS-Aufrufe }
  CmdStr: string[3];
  Off: Boolean;             { Flag für "Parameter OFF" }
  Installed: Boolean;       { Flag für "bereits installiert" }
  Ch: Char; x:Integer;
  EnvSeg: Word;             { Environment-Segmentadresse }
  ResData: Word;            { für die Daten-Segmentadresse des
                              residenten Programms }
  p: ^Word;                 { für diverse Zwecke }
  pp: ^Pointer;             { für den 2. INT-Vektor }
  IDPtr: ^ThreeIDs;         { für die Kennbytes }
```

Diese Variablen sind sämtlich lokal deklariert und werden deshalb auf dem Stack gespeichert. Der Grund für diese massierte Deklaration und die Überlänge von *IntCheck*: Wir sparen auf diese Weise Platz im Datensegment (und damit im Hauptspeicher, wenn das Programm installiert ist).

```
begin
  TurboSS := SSeg; TurboSP := StackTop;   { *** !!! *** }

  { 1. Schritt: Kommandozeilen-Parameter "Off"? }
  if ParamCount >= 1 then
  begin
    Off := True;
    CmdStr:= Copy(ParamStr(1),1,3);
    for x:= 1 to 3 do
      if UpCase(CmdStr[x]) <> OFFCMD[x] then Off:= False;
    if not Off then
    begin
      Writeln('Ungültiger Kommandozeilen-Parameter: ', ParamStr(1));
      IntCheck:= False; Exit;
     end;
   end
  else Off := False;
```

Vor dem »1. Schritt« mit der Prüfung auf den Kommandozeilen-Parameter OFF werden die Variablen *TurboSS* und *TurboSP* mit der Adresse des Turbo-Stacks gesetzt. Auf diese Werte greift das Assemblerprogramm *IntFrame* beim Umsetzen des Stacks zu.

```
  { 2. Schritt: Prüfung, ob das Programm bereits installiert wurde.
    Wenn ja:  a) Kommandozeilen-Parameter "Off" -> hinauswerfen
              b) ansonsten Meldung "bereits installiert" und Abbruch }
  Installed := False;
  GetIntVec(IntNo,SaveIntVec);
  if (PWord(SaveIntVec).Off = Ofs(IntFrame))
    and (SaveIntVec <> nil) then
  begin
    { Ermittlung des Datensegments des residenten Programms }
    ResData := PWord(SaveIntVec).Seg     { CSeg des residenten Programms }
               + (DSeg - Seg(IntFrame)); { plus Abstand CSeg | DSeg }

    Installed := True;
    IDPtr := Ptr(ResData,Ofs(IDs));
    for x:= 0 to 2 do
      if IDPtr^[x] <> IDs[x] then Installed := False;

    { Diese Daten werden auch dann (unnötigerweise) ermittelt,
      wenn das ID nicht übereinstimmt. }
    p := Ptr(ResData,Ofs(OrgIntOff));   { Original-Interrupt: Offset }
    OrgIntOff := p^;        { wird in unserem OrgIntOff gespeichert }
    p := Ptr(ResData,Ofs(OrgIntSeg));   { Segment-Adresse }
    OrgIntSeg := p^;
    p := Ptr(ResData,Ofs(Prefix));      { Programmsegment-Präfix }
    Prefix := p^;
```

```
    EnvSeg := Word(Ptr(Prefix,$2C)^);    { Environment }
    pp := Ptr(ResData,Ofs(SaveInt2Vec));
    SaveInt2Vec := pp^;                  { 2. Interruptvektor }
  end;
```

Wenn der Offset-Anteil des Interruptvektors und der Offset-Anteil von *IntFrame* übereinstimmen, dann ermittelt *IntCheck* das (vermutliche) Datensegment des residenten Programms, setzt den Zeiger *IDPtr* auf die Adresse, ab der sich die Kennbytes befinden sollten, und vergleicht die gespeicherten Daten.

Die restlichen Eckdaten des residenten Programms werden nach demselben Verfahren ermittelt, d.h. über *ResDSeg* und die jeweiligen Offset-Adressen des laufenden Programms. Damit hier nicht noch ein weiterer if-Zweig fällig ist, geschieht diese Ermittlung auch dann, wenn *Installed* bereits den Wert *False* angenommen hat.

Der »2. Interruptvektor« ist in PRTFILE der Vektor zum BIOS-Interrupt $17: Der Originalwert dieses Vektors wird in der Variablen *SaveInt2Vec* gespeichert, damit auch dieser Vektor bei einer eventuellen Entfernung eines eingelagerten Programms wieder zurückgesetzt werden kann. Die Variable *pp* ist ausschließlich für das Lesen dieses Vektors deklariert, weil der Compiler eine Typ-Umwandlung der Form `Pointer(p^)` zwar widerspruchslos schluckt - das Programm würde in diesem Fall aber nur den Offset-Anteil des Vektors lesen (siehe auch Anhang D).

```
  if (not Installed) and Off then
  begin
    Writeln('Programm nicht installiert/nicht als erstes in der Kette.');
    IntCheck := False;
    Exit;
  end;
  if Installed then
  begin
    if not Off then
    begin
      Writeln('Programm ist bereits installiert.');
      IntCheck:= False;
      Exit;
    end
    else begin
      with Regs do
      begin
        AH := $49;        { DOS-Funktion: Free Memory (Segment in ES }
        ES := Prefix;
        Intr($21,Regs);   { Freigabe des Programms }
        AH := $49;
        ES := EnvSeg;
        Intr($21,Regs);   { Freigabe des Environments }

        p:= Ptr(OrgIntSeg,OrgIntOff);
```

```
        { Und schließlich: Zurücksetzen der Interrupts }
        SetIntVec(IntNo,p);
        SetIntVec(Int2No,SaveInt2Vec);
      end;  { with Regs }
     Writeln('Programm aus dem Speicher entfernt');
     Halt(0);
    end; { else }
  end; { if Installed }

  IntCheck:= True;
end;
```

Wenn der Kommandozeilen-Parameter OFF angegeben wurde und *IntCheck* eine eingelagerte Programmkopie erkannt hat, wird dieses Programm aus dem Speicher entfernt: Über einen Aufruf der DOS-Funktion $49 (»Free Allocated Memory«) werden zuerst PSP-Segment und Programm freigegeben, danach die dazugehörige *Environment*-Tabelle. Das Register ES muß bei diesen beiden Aufrufen jeweils die Segmentadresse des freizugebenden Speicherblocks enthalten.

Schließlich setzt *IntCheck* die beiden durch das eingelagerte Programm veränderten Interruptvektoren (für PRTFILE: $21 und $17) wieder auf ihre Originalwerte zurück.

Licht ins Dunkel des Coprozessors: TRACE287

Haben Sie schon einmal das Vergnügen gehabt, bei der direkten Programmierung des Coprozessors einen Ablauf schrittweise mit DEBUG, SYMDEB, Periscope etc. zu verfolgen und die Register des 8087 dabei mit Papier und Bleistift zu »beobachten«? Alle mir bekannten Debugger*) liefern zwar ausführliche Informationen über die Register des 8086 - was den Coprozessor anbelangt, ist man auf die zuvor erwähnten Hilfsmittel und viel Optimismus angewiesen.

TRACE287.PAS demonstriert zwei Dinge:

- Eine weitere Anwendungsmöglichkeit von DOSHOOK. Der »abzufangende BIOS-Interrupt« ist in diesem Fall die Print Screen-Funktion und wird zur Umschaltung zwischen »Anzeige an« und »Anzeige aus« benutzt. Die Prozedur *PasProc* wird bei jedem Aufruf der DOS-Funktion $0A aktiviert (d.h. nach jeder Ausführung eines Befehls von DEBUG oder SYMDEB) - sie gibt die Registerinhalte des Coprozessors in einem eigenen »Fenster« aus, wenn »Anzeige an« gesetzt ist.
- Den Grund, warum ich wohl nie richtig reich werde. Jeder vernünftige Mensch würde TRACE287 auf einer Diskette zusammen mit einigen Seiten hübsch

*) Wohlgemerkt: Debugger. Fehlervernichtungsmonstren wie der Turbo Debugger, CodeView 5.0 und andere Programm-WAAs ähnlicher Größenordnung sind hier nicht berücksichtigt.

gemachter Dokumentation (und ohne die Quelltexte) verkaufen - ungefähr für den Preis dieses Buches.

Das Programm besteht aus drei Dateien: DOSHOOK.ASM (Listing und Besprechung auf Seite 264), INTCHK2.INC (siehe vorhergehender Abschnitt) und TRACE287.PAS.

```
program Trace287;   { TRACE287.PAS 12/87, 12/88 as }
{ Ausgabe der Register des Coprozessors bei jedem Aufruf der DOS-
  Funktion $0A (Buffered Keyboard Input).
  Arbeitet nach demselben Schema wie HOOKDEMO und PRTFILE,
  d.h. benutzt DOSHOOK.ASM und INT2CHK.INC.

  Das Programm läßt sich (mit TRACE287 OFF)
  wieder aus dem Speicher entfernen.
}

{$M 2000,0,0}              { Achtung: 2000 Bytes Stack }
{$S+,R-,I-,N+}             { Coprozessor vorausgesetzt! }
{$IFDEF VER50}
  {$N+,E-}       { <- läuft *nicht* mit dem Emulator! }
{$ENDIF}
```

TRACE287 setzt auch in der Version 5.0 einen installierten Coprozessor voraus, weil das Programm sonst fast ein Dutzend Interruptvektoren blockieren müßte (siehe Kapitel 2). Der dafür verwendete Befehl zur bedingten Compilierung bringt uns zu einer Kuriosität, die thematisch überhaupt nicht an diese Stelle gehört: Eigentlich sollte man meinen, daß sich ein in der Version 4.0 nicht definierter Befehl wie {$E-} mit einem Konstrukt à la {$IFDEF VER50} {$E-} {$ENDIF} von der Compilierung ausschließen läßt. Tatsächlich ist dem nicht so: nachdem der Compiler den {$IFDEF}-Befehl als False bewertet hat, sucht er nach dem dazugehörigen {$ENDIF}. Die Suche endet beim nächsten '{$' im Quelltext - und mit der (durchaus richtigen) Feststellung, daß die darauffolgenden Zeichen (E-) einen unbekannten Befehl darstellen. Wenn wir dem Compiler dagegen als erstes einen bekannten Befehl (wie eben {$N+,...}) servieren, dann kann er die Analyse bis zur Erkenntnis abschließen, daß es sich bei diesem Befehl nicht um ein {$ENDIF} handelt - woraufhin er den folgenden Text bis zum nächsten '{$' ignoriert.

Zurück zum Listing von TRACE287:

```
uses Crt, Dos;    { wg. GotoXY bzw. Registers, SetIntVec usw. }

type
  PWord = record   { Typ-Umwandlung für Zeigerwerte }
     Off: Word;       { Offset }
     Seg: Word;       { Segment }
   end;
  ThreeIDs = array[0..2] of Byte; { Typ-Umwandlung für IDs }
```

```
const
  IDs: ThreeIDs = (22,12,87);    { ID des Programms }
  OrgIntOff: Word = 0;           { Hier wird der Original- }
  OrgIntSeg: Word = 0;           { Interrupt-Vektor gesetzt }
  Prefix: Word = 0;              { und hier das Programmsegment-Präfix }

  FUNCNO:Byte = $0A;  { Bei dieser DOS-Funktion wird DOSPROC eingeschoben }
  DOSINT = $21;     { Dieser Interrupt wird ersetzt }
  PRTSCINT = $05;  { und der auch }
```

Der Print Screen-Interrupt ($05) wird zur Umschaltung der Anzeige verwendet und komplett ersetzt. Jeder Aufruf der DOS-Funktion $0A (»Buffered Keyboard Input«) aktiviert die Prozedur *PasProc*; die DOS-Funktion selbst bleibt unverändert und wird nach dem Aufruf von *PasProc* ausgeführt.

```
var
  TurboSS, TurboSP: Word;  { SS:SP von Turbo }
  SaveIntVec: Pointer;     { zum Speichern des Originalvektors von DOS }
  SaveInt2Vec: Pointer;    { für den Originalvektor von Shift-PrtSc }
  p: ^Pointer;

  Active: Boolean;         { Flag für "Anzeige an/aus" }
```

In den Variablen *TurboSS* und *TurboSP* werden SS:SP des Pascal-Programms gespeichert. Das Assemblerprogramm *IntFrame* liest diese beiden Werte und setzt darüber den Stack des Pascal-Programms vor dem Aufruf von *PasProc*. *SaveIntVec* speichert den Originalvektor zu DOS, *SaveInt2Vec* den Originalvektor des Print Screen-Interrupts. Diese beiden Werte werden nur verwendet, wenn das Programm wieder aus dem Speicher entfernt werden soll.

Die Variable *Active* wird über die **interrupt**-Prozedur *ActiveOnOff* bei jeder Eingabe von SHIFT-PRTSC negiert - ihr momentaner Wert bestimmt, ob *PasProc* die Register des Coprozessors ausgibt oder nicht.

```
procedure IntFrame; external;
procedure DosOrg; external;      { muß auf den DOS-Originalvektor gesetzt
                                   werden }
{$L DOSHOOK}
```

Der DOS-Interruptvektor wird vom Hauptprogramm auf *IntFrame* gesetzt - *IntFrame* prüft bei jedem DOS-Aufruf die dazugehörige Funktionsnummer. Wenn es sich um die Funktion FUNCNO (= $0A) handelt, wird die Pascal-Prozedur *PasProc* vor der Ausführung der DOS-Funktion eingeschoben.

DosOrg ist nur formal als Prozedur deklariert und steht für vier Speicherzellen im CODE-Segment von *IntFrame*, in die das Programm bei seiner Installation den Originalvektor zu DOS hineinschreibt.

```
const
  VLINE = #186;   { Grafikzeichen: senkrechter Doppelstrich }
  HLINE = #205;   {    ""        : waagrechter Doppelstrich }
  EDGE  = #200;   {    ""        : Ecke unten links         }

  { Exception-Flags des Coprozessors:
    Invalid, Denorm, Zero, Overflow, Underflow, Precision }
  ExFlags : array[0..5] of Char = ('I', 'D', 'Z', 'O', 'U', 'P');
  { Tags der Register des Coprozessors:
    Valid, Zero, Special (NaN, Denorm, Inf), Empty }
  TagFlags : array[0..3] of Char = ('V', 'Z', 'S', 'E');

var
  ST87 : record
           Control: Word;   { x  x  x  IC RC RC PC PC x  x PM UM OM ZM DM IM }
           Status : Word;   { B  C3 ST ST ST C2 C1 C0 ES x PE UE OE ZE DE IE }
           Tags   : Word;   { R7 R7 R6 R6 R5 R5 .... R2 R2 R1 R1 R0 R0 }
           IPOff,IPHi,       { absolute Adresse des letzten Befehls }
           DTOff,DTHi: Word; { absolute Adresse des letzten Datenzugriffs }
           CRegs  : array[0..7] of Extended;  { die einzelnen Register }
         end;
```

In dieser Form speichert der Coprozessor seine Status- und Rechenregister, wenn der Befehl FSAVE gegeben wird. Die Felder *IPOff..DTHi* enthalten die absolute Adresse des letzten 8087-Befehls *vor* FNSAVE zusammen mit dem Befehlscode bzw. die absolute Adresse des letzten Datenzugriffs durch den 8087 (jeweils 19 Bit) und sind für TRACE287 nicht weiter von Interesse. Eingehendere Erläuterungen der einzelnen Flags und Datentypen wie *Denorm*, *NaN* usw. finden Sie in Kapitel 2.

```
var   VStr: string;            { für Str, Grafikzeichen ... }

procedure RealToVStr(Num:Extended; Decs:Integer);
begin
  Str(Num:Decs, VStr);
end;
```

RealToVStr speichert eine Fließkommazahl als ASCII-Folge in der globalen Variablen *VStr* und ist nur wegen des Emulators der Version 5.0 als eigene Routine definiert.

```
{ ** Die Interrupt-Prozedur ** }
procedure PasProc;
var
  Stack87: Integer;         { Stackzeiger des 8087 }
  x, CurrentReg: Integer;
  RegState: Byte;
  CursorX, CursorY: Byte;   { originale Cursorposition }
```

```
begin
  if not Active then Exit;  { Ausgabe nur dann, wenn das Flag
                              "Active" (via Shift-PrtSc) gesetzt ist }
  CursorX := WhereX;
  CursorY := WhereY;       { Original-Cursorposition festhalten }

  inline($9B/              { WAIT         ; falls der Coproc beschäftigt ist }
         $DD/$36/ST87/     { FSAVE ST87   ; siehe oben }
         $9B/              { WAIT         ; warten auf das Ende
                                            der Speicherung! }
         $DB/$E3/          { FINIT }
         $9B);             { WAIT }
```

Der Beginn von *PasProc*: Wenn das Flag *Active* gesetzt ist, wird die momentane Position des Cursors festgehalten, danach folgen die Anweisungen für den Coprozessor, seinen Zustand in der globalen Variablen ST87 zu speichern. Es ist zwar recht unwahrscheinlich, daß der 8087 noch mit einer Rechenaufgabe beschäftigt ist, während das Programm bereits den nächsten DOS-Aufruf durchführt; die halbe Mikrosekunde für den WAIT-Befehl läßt sich aber sicher verschmerzen. Nach FSAVE muß dagegen auf jeden Fall ein WAIT-Befehl stehen - immerhin sind hier 94 Bytes auf einmal zu schreiben.

Das folgende FINIT initialisiert den 8087 und könnte unnötig sein - bei der Frage, ob ein FSAVE gleichzeitig eine Initialisierung durchführt, scheiden sich die Geister der Autoren.

```
with ST87 do
begin
  Stack87 := Status shr 11 and $07;   { Bits 11/12/13: 8087-Stackzeiger }

  for x:= 7 downto 0 do
  begin
    CurrentReg := (8 + x - Stack87) mod 8;  { "8+" sorgt für > 0 }
    RegState := Tags shr (x * 2) and $03;
    if RegState = 3                     { Register ist "Empty" }
      then VStr := ' - - - - - - - - '      { <- 17 Zeichen }
      else RealToVStr(CRegs[CurrentReg],17); { Reg-Inhalt -> VStr }
    if x = Stack87
      then VStr := VLINE + ' S>'                  { Stack-Markierung }
                 + VStr + ' ' + TagFlags[RegState] + ' '
      else VStr := VLINE + ' '+ chr(x + ord('0'))  { sonst Reg-Nummer }
                 + ' ' + VStr + ' ' + TagFlags[RegState] + ' ';

    GotoXY(56,8-x); Write(VStr);            { Ausgabe }
  end; { for x }
```

Die Register werden in ihrer physikalischen Reihenfolge ausgegeben (d.h. unabhängig vom momentanen Stand des Stackzeigers steht R7 immer zuoberst). Da der Befehl FNSAVE die Registerinhalte aber mit dem TOS zuerst speichert (*CRegs[0]*

enthält also immer das TOS, *CRegs[1]* das Register ST(1) usw.), müssen wir die Indexe in *CRegs* über den momentanen Wert des Stackzeigers umrechnen.

Der Zustand jedes Rechenregisters ist mit zwei Bits im Tag-Register codiert (jeweils 4 mögliche Werte: Gültig, Null, »Speziell« oder Leer). Im Gegensatz zu den sonstigen Operationen des 8087 ist diese Zuordnung vom momentanen Stand des Stackzeigers unabhängig (Bits 15..14: R7, Bits 13..12: R6 usw).

```
    { Ausgabe der Statusbits:
      VStr:=   '|   P U O Z D I  IC Rx Px' }

    FillChar(VStr[1],24,' ');  VStr[0] := #24; { 24 Leerzeichen }
    VStr[1] := VLINE;

    for x:= 5 downto 0 do          { Einsetzen der Flags }
      if Status shr x and $01 <> 0     { Flag gesetzt }
        then VStr[(5-x) * 2 + 4] := ExFlags[x];
    if Control and $1000 <> 0 then
    begin
      VStr[17]:= 'I'; VStr[18] := 'C';
     end;
    VStr[20] := 'R';    { Rundungsrichtung: Bits 10/11 im Statuswort }
    VStr[21] := chr((Control shr 10) and $03 + ord('0'));
    VStr[23] := 'P';    { Genauigkeit: Bits 8/9 im Statuswort }
    VStr[24] := chr((Control shr 8) and $03 + ord('0'));
    GotoXY(56,9); Write(VStr);  { Ausgabe }

    { unterste Zeile: VStr := '|----------------' }
    VStr[1] := EDGE;
    FillChar(VStr[2],23,HLINE);
    GotoXY(56,10); Write(VStr); { Ausgabe }
  end; { with ST87 }
```

Die Ausgabe der Statusflags (auch hierzu - siehe Kapitel 2): P..I werden durch die jeweils letzte vorherige Operation beeinflußt. Sie stehen für »Precision« (= Ungenau), »Underflow«, »Overflow«, »Zero« (= Division durch 0), »Denorm« und »Invalid«.

```
  inline($9B/                { WAIT          ; sehr unwahrscheinlich, aber... }
        $DD/$26/ST87/        { FRSTOR ST87   ; Zurückspeichern }
        $9B);                { WAIT          ; warten auf das Ende
                                               der Aktion }
  GotoXY(CursorX, CursorY);  { Cursor zurück }
end;
```

Unabhängig davon, welche Veränderungen der FSAVE-Befehl am Anfang der Routine nun bewirkt hat - die Umwandlung der Felder *CRegs* (Registerwerte) von *ST87* in Strings (mit *Str*) hat den 8087 auf jeden Fall beschäftigt. Erfreulicherweise läßt sich sein ursprünglicher Zustand durch das Zurückladen der in *ST87* gespei-

cherten Daten bis aufs letzte Bit wiederherstellen. (Die WAIT-Befehle sind hier wahrscheinlich ebenfalls überflüssig, weil als nächstes erst einmal die DOS-Funktion zum Lesen einer Eingabezeile durchgeführt wird - trotzdem sollte man sie lassen, wo sie sind).

```
procedure ActiveOnOff; interrupt;
begin
  Active := not Active;
end;
```

Diese Prozedur wird über SHIFT-PRTSC aufgerufen und tut nichts weiter, als den Zustand des Flags »Active« umzudrehen, d.h. zwischen »Anzeige an« und »Anzeige aus« umzuschalten.

Wenn *Active* auf *True* gesetzt ist, gibt *PasProc* bei jeder Aktivierung die Registerinhalte des Coprozessors neu aus. Für *Active* = *False* unterbleibt diese Ausgabe einfach - mit der Folge, daß die angezeigten Registerwerte über kurz oder lang nach oben aus dem Bildschirm gerollt werden.

```
{$I INT2CHK.INC}     { Prüfung auf Kommandozeilen-Parameter OFF,
                       bereits installiertes Programm usw. }
```

Weil TRACE287 zwei Interruptvektoren auf einmal belegt, wird hier die Version der Funktion *IntCheck* mit drei Parametern verwendet - Listing und Erläuterungen finden Sie auf Seite 276.

```
{ ------------------------------------------------------------ }
begin
  { *** Achtung: Aus Gründen, die ich auch nicht weiter erklären
    kann oder will, formatiert das Programm die Festplatte neu und
    gießt Erdbeermarmelade in die Tastatur, wenn der folgende
    Aufruf von Writeln nicht exakt in der gezeigten Form
    programmiert wird. }

  Writeln('TRACE287 - (c) as 12/87, 12/88');

  if ParamStr(1) = '?' then     { Eine halbwegs anständige Hilfestellung }
  begin
    Writeln('Aufruf: TRACE287 [OFF]');
    Writeln('Anzeige der 80x87-Register, Umschaltung mit Shift-PrtSc');
    Writeln('Der Aufruf mit OFF entfernt TRACE287 wieder aus dem Speicher.');
    Halt(1);
  end;
```

So ungefähr sollte eigentlich jedes verwendbare Programm auf den Start mit einem Fragezeichen als Kommandozeilen-Parameter reagieren. Zugegeben ist es etwas lästig, sich seine eigenen Hilfestellungen zu schreiben, aber: Jeder, der mehr als ein paar Dutzend selbstgeschriebene Programme auf der Festplatte hat, von denen einige nur recht selten verwendet werden, ist sich irgendwann einmal für diese Mühe dankbar.

Was das »Achtung« betrifft: Selbstverständlich sind alle Programme dieses Buches zur schrankenlosen Weiterverbreitung freigegeben - unsereins lebt aber nun einmal davon, daß der eigene Name auf anderer Leute Bildschirm erscheint und sie neugierig macht*).

```
if not IntCheck(DOSINT, PRTSCINT,SPtr) then Halt(1);

GetIntVec(DOSINT,SaveIntVec);  { Originalwert des DOS-Vektors }

OrgIntOff := PWord(SaveIntVec).Off;
OrgIntSeg := PWord(SaveIntVec).Seg;
Prefix := PrefixSeg;

GetIntVec(PRTSCINT,SaveInt2Vec);  { Originalwert des PrintScreen-Vektors }
```

Die Funktion *IntCheck* prüft, ob sich bereits eine Kopie des Programms im Speicher befindet und entfernt sie, wenn der Kommandozeilen-Parameter OFF angegeben wurde bzw. verhindert eine mehrfache Installation desselben Programms. Das Funktionsergebnis von *IntCheck* ist nur dann *True*, wenn das Programm tatsächlich in den Speicher eingelagert werden soll.

Die Speicherung der Eckdaten vor der Installation: *OrgIntOff*, *OrgIntSeg* und *SaveInt2Vec* ermöglichen es, die Originalvektoren bei einer späteren Entfernung des Programms wieder zu setzen; über die in *Prefix* gespeicherte Segmentadresse wird in diesem Fall der vom Programm belegte Speicherblock wieder freigegeben.

```
{$IFDEF VER40}
  SetIntVec(0,SaveInt00);          { Dieser Interrupt ist noch von SYSTEM
                                     belegt und wird nun hinausgeworfen }
  SetIntVec($75,SaveInt75);        { Interrupt für Coproc-Fehler }
  SetIntVec($02,SaveInt02);        { dito }
  SetIntVec($1B,SaveInt1B);        { ^Break-Vektor von CRT zurück }
{$ELSE}
  RealToVStr(Pi, 17); { Umwandlung der EMU-Befehle von
                        RealToVStr in 8087-Ops   *** UNBEDINGT NOTWENDIG! }
  SwapVectors; { Vektoren 0,2,1B,34..3E,3F, 75 zurück }
{$ENDIF}

  CheckBreak := False;
  DirectVideo := True;             { für Grafik: auf FALSE setzen }

  Writeln('Programm installiert');
  Active := True;                  { damit man auch etwas zu sehen bekommt }
```

Da das Programm mit dem Compiler-Befehl {$N+} compiliert wurde und darüber hinaus noch das Unit *Crt* verwendet, muß hier eine ganze Reihe von Interrupt-

*) Ganz ehrlich: Es ist immer wieder erstaunlich, wieviele Leute aus zwei- oder dreihundert Zeilen Quelltext eine einzige weglassen und dann meinen, sie hätten das Programm selbst geschrieben.

Vektoren wieder zurückgesetzt werden (vgl. »Die Initialisierungsteile der Standard-Units« in Kapitel 1 und »Die Initialisierung des 8087« in Kapitel 2).

Diese Aktion läßt sich in der Version 5.0 auf einen Streich erledigen - dort kommt aber noch etwas anderes hinzu: Sowohl die Bibliotheksroutine *Str* als auch *die Parameter-Übergabe zu ihrem Aufruf* enthalten Coprozessor-Befehle, die in dieser Compilerversion als $3x-Interrupts codiert sind und deshalb einmal durchlaufen werden müssen, damit der Emulator echte 8087-Opcodes daraus macht. Was uns zu einem »Vorsicht« bringt, das nur für die Version 5.0 und ihre (hypothetischen) Nachfolger gilt:

VORSICHT mit der Version 5.0: Wir haben hier das unverschämte Glück, daß TRACE287 a) nur eine einzige Fließkomma-Routine benutzt, die sich zusammen mit ihrem Aufruf als eigene Routine (*RealToVStr*) problemlos auslagern läßt und b) *Str* zumindest in der .0-Version der 5er-Generation noch keinen der »Shortcuts« (INT $3E) verwendet. Oder auch: Durch den Aufruf von *SwapVectors* wird nicht nur dem Emulator, sondern auch der Fließkomma-Fehlerbehandlung der Boden unter den Füßen weggezogen. Letztere enthält definitiv zwei »Emu-Interrupts« und rächt sich nur deshalb nicht fürchterlich, weil das FINIT zu Beginn von *PasProc* sämtliche Ausnahmezustände des 8087 maskiert...

Deshalb eine herzliche Bitte: spendieren Sie also jedem eigenen Programm, in dem Sie die hier demonstrierte Wildwestmethode einsetzen wollen, eine extra Gedenkminute.

Zurück in zivilisierte Gebiete: Solange die Variable *DirectVideo* auf *True* gesetzt ist, schreiben Aufrufe von *Writeln* direkt in den Bildspeicher, und das Programm benötigt ungefähr die Hälfte der Zeit zur Ausgabe des »Fensters«. Wenn das mit TRACE287 überwachte Programm allerdings einen Grafikmodus setzt, bekommen Sie nur noch ein paar bunte Klötzchen zu sehen, weil *Crt* diese Umschaltung naturgemäß nicht mitbekommt. Für *DirectVideo* = *False* verwendet *Writeln* dagegen einen (langsameren) BIOS-Aufruf - der funktioniert aber auch in der Grafik.

```
{ *** Achtung: Die Installation muß der letzte Befehl vor KEEP sein -
  ansonsten könnte sich das Programm selbst unterbrechen! }

  p := @DosOrg; p^:=  SaveIntVec;     { DOS-Vektor für IntFrame }

  SetIntVec(PRTSCINT,@ActiveOnOff);  { Installation von ActiveOnOff }
  SetIntVec(DOSINT,@IntFrame);       { Installation von IntFrame/PasProc }
  Keep(0);  { TSR, Exit-Code 0 }
end.
```

Da die Aktivierung von *PasProc* von einem DOS-Aufruf abhängt (und nicht vom »Clock Tick« der eingebauten Uhr), ist der Zeitpunkt der endgültigen Veränderung der Interrupt-Vektoren nicht sonderlich kritisch - die einzige Möglichkeit, mit Sicherheit einen Systemabsturz herbeizuführen, besteht in einem Aufruf von *Readln* nach dem Setzen von DOSINT.

Ersatz von DOS-Funktionen

Wenn wir dem Betriebssystem mit einem Programm wie DOSHOOK.ASM über die Schulter schauen können, dann müßte es doch auch möglich sein, ihm mit einer leichten Modifikation in die Suppe zu... ? Leider eignet sich diese Assoziation nicht sonderlich für Überschriften*).

Das zugrundeliegende Funktionsprinzip unterscheidet sich von DOSHOOK.ASM tatsächlich nur in einem Detail: Für eine bestimmte DOS-Funktion wird nicht eine Pascal-Routine *eingeschoben*, sondern *anstelle dieser Funktion ausgeführt*. Da DOS-Aufrufe aber im allgemeinen Ergebnisse in Form gesetzter Register zurückliefern und wir bei DOSHOOK bewußt auf die Übergabe der entsprechenden Werte verzichtet haben, fallen die Veränderungen doch etwas größer aus.

Das Assemblerprogramm DOSCATCH.ASM ist eine Kombination aus DOSHOOK.ASM (siehe Seite 264) und INTFRAME.ASM - dem allerersten »Rahmenprogramm«, das in diesem Kapitel besprochen wurde (Seite 237).

```
; DOSCATCH.ASM { 22/12/87 as }
; Ersatz einer DOS-Funktion durch eine Pascal-Routine

Registers STRUC
rAX       dw ?          ; Struktur des Turbo Pascal-Datentyps "Registers"
rBX       dw ?
rCX       dw ?
rDX       dw ?
rBP       dw ?
rSI       dw ?
rDI       dw ?
rDS       dw ?
rES       dw ?
rFlags    dw ?
Registers ends

DosVec  equ 0084h                ; Adresse des DOS-Interruptvektors

DATA    Segment Word PUBLIC
        EXTRN Regs:Word                    ; Registers-Record
        EXTRN SSSave:Word,SPSave:Word      ; zum Speichern von SS:SP
        EXTRN CallCS:Word,CallIP:Word      ; zum Speichern von CS:IP
        EXTRN TurboSS:Word,TurboSP:Word    ; SS:SP von Turbo
        EXTRN FuncNo:Byte                  ; Nummer der DOS-Funktion
DATA    ends
```

Auch die Variablen zur Zwischenspeicherung von CS:IP und SS:SP des aufrufenden Programms sind im Pascal-Programm deklariert und damit für *PasProc* ver-

*) Oder auch: DOS eins über die Rübe gehauen...

fügbar. Gegen eine schlichte Ausgabe von CS:IP oder SS:SP wie mit TRACE86R ist nichts einzuwenden - wer diese Werte allerdings *verändert*, tut das auf eigene Gefahr.

```
CODE    Segment Byte PUBLIC
        PUBLIC IntFrame
        PUBLIC DosOrg           ; originaler DOS-Interruptvektor
        EXTRN  PasProc:near     ; die aufzurufende Pascal-Prozedur
        ASSUME cs:CODE, ds:DATA

IntData dw DATA                 ; Turbo-Datensegment
DosOrg  dw 0,0                  ; Originalvektor zu DOS, von Turbo gesetzt
IntFrame proc FAR
        push ds
        mov ds,IntData          ; DS aufs Turbo-Datensegment
        cmp ah,FUNCNO           ; sollen wir aktiv werden?
        jz CallTurbo
        pop ds
        push [DosOrg+2]
        push [DosOrg]
        ret                     ; nein.. ab zu DOS
```

Schritt 1: Die Prüfung, ob »unsere« DOS-Funktion aufgerufen wurde. Wenn nein, geht es mit einem direkten Sprung zu DOS weiter, der hier wegen der Probleme mit dem Linker wieder als RET-Befehl ausgeführt wird (vgl. Anhang D).

```
CallTurbo: pop [Regs.rDS]       ; Stack (von oben nach unten): Flags
        pop [CallIP]            ;          Rücksprungadresse   CS
        pop [CallCS]            ;                              IP
        pop [Regs.rFlags]       ;                              DS
        mov [SSSave],ss         ; SS:SP sind um sechs erhöht
        mov [SPSave],sp         ; (wg. CS:IP, Flags)
        mov [Regs.rAX],ax
        mov [Regs.rBX],bx
        mov [Regs.rCX],cx
        mov [Regs.rDX],dx       ; die restlichen Registerwerte
        mov [Regs.rBP],bp       ; in die Turbo-Variable REGS
        mov [Regs.rSI],si
        mov [Regs.rDI],di
        mov [Regs.rES],es

        cli                     ; Stack von Turbo setzen:
        mov ss,[TurboSS]        ; hier dürfen wir von niemanden
        mov sp,[TurboSP]        ; unterbrochen werden
        sti
```

Schritt 2: Speichern aller Registerinhalte in der Pascal-Variablen *Regs* und Umsetzen des Stacks.

```
        xor ax,ax
        mov es,ax               ; ES auf Segment 0000
        push es:[DosVec]        ; momentanen INT 21-Vektor
        push es:[DosVec+2]      ; auf den Stack
        mov ax,[DosOrg]
        mov es:[DosVec],ax      ; DOS-Interruptvektor auf DOS
        mov ax,[DosOrg+2]
        mov es:[DosVec+2],ax
```

Vor dem Aufruf der Routine *PasProc* wird (wie in DOSHOOK.ASM) der Originalvektor zu DOS gesetzt - DOS-Aufrufe innerhalb von *PasProc* führen also tatsächlich zu DOS (oder zuerst zu einem weiteren TSR-Programm, das diesen »Originalvektor« besetzt hält, vgl. Seite 261).

```
        call PASPROC            ; Aufruf der Pascal-Routine


        xor ax,ax
        mov es,ax               ; ES wieder auf Segment 0000
        pop es:[DosVec+2]       ; INT 21-Vektor wieder zurück
        pop es:[DosVec]


        mov es,[Regs.rES]
        mov bp,[Regs.rBP]
        mov di,[Regs.rDI]       ; Felder von REGS zurück in die
        mov si,[Regs.rSI]       ; Prozessor-Register
        mov dx,[Regs.rDX]
        mov cx,[Regs.rCX]
        mov bx,[Regs.rBX]
        mov ax,[Regs.rAX]       ; inklusive AX

        cli
        mov ss,[SSSave]         ; SS:SP zurück aufs Original
        mov sp,[SPSave]
        sti
```

Nach der Ausführung von *PasProc* wird der DOS-Vektor wieder auf das Programm zurückgesetzt, darauf folgt die Rückspeicherung der Register und das Wiederherstellen des Original-Stacks.

```
        push[Regs.rFlags]       ; Flags und Rücksprungadresse
        push[CallCS]            ; auf den Stack des aufrufenden
        push[CallIP]            ; Programms
        push[Regs.rDS]
        pop ds                  ; DS zurück aufs Original
        iret
IntFrame endp
CODE    ends
        end
```

CATCH22...

ist vorgeblich als Test für die Funktionsfähigkeit von DOSCATCH.ASM gedacht. Das Programm ersetzt die DOS-Funktion $3A durch eine Routine mit eigenen Ansichten über das Löschen von Directories.

```
program Catch22;  { 22/12/87 as }
{ Demonstration von DOSCATCH.ASM: Die Pascal-Routine DOSPROC ersetzt
  die Funktion "Remove Directory" (AH = $3A).
}

{$M 2000,0,0}              { *** HeapMax = 0 *** }
{$S+,R-,I-}

uses Crt,Dos;    { wg. GotoXY bzw. Registers, SetIntVec usw. }

type
  PWord = record   { Typ-Umwandlung für Zeigerwerte }
     Off: Word;       { Offset }
     Seg: Word;       { Segment }
   end;

  ThreeIDs = array[0..2] of Byte; { Typ-Umwandlung für IDs }

const
  IDs: ThreeIDs = (22,12,87);   { ID des Programms }
  OrgIntOff: Word = 0;          { Hier wird der Original- }
  OrgIntSeg: Word = 0;          { Interrupt-Vektor gesetzt }
  Prefix: Word = 0;             { und hier das Programmsegment-Präfix }

  FUNCNO:Byte = $3A;  { Diese Funktion wird durch PASPROC ersetzt }
  DOSINT = $21;  { Dieser Interrupt wird ersetzt }
```

Die globalen Deklarationen sind im Prinzip dieselben wie bei den zuvor besprochenen TSR-Programmen. FUNCNO muß als typisierte Konstante deklariert sein, damit das Assemblerprogramm darauf zugreifen kann.

```
var
  TurboSS, TurboSP: Word;  { SS:SP von Turbo }
  SaveIntVec: Pointer;     { zum Speichern des Originalvektors }
  p: ^Pointer;
  Regs: Registers;         { Übergabe der Registerwerte }
  CallCS, CallIP: Word;    { CS:IP des aufrufenden Programms }
  SSSave, SPSave: Word;    { SS:SP des aufrufenden Programms }

procedure IntFrame; external;
procedure DosOrg; external;     { muß auf den DOS-Originalvektor gesetzt
                                  werden }
{$L DOSCATCH}
```

Da wir CATCH22 (im Gegensatz zu TRACE287) mit Informationen über die Registerwerte des aufrufenden Programms versorgen müssen, sind hier die Variablen *Regs*, *CallCS* und *CallIP* wieder im Pascal-Programm deklariert.

```
procedure PasProc;   { diese Prozedur ersetzt eine DOS-Funktion }
var p: ^string;
    DirName: string;
    x,y,Color: Integer;
begin
   with Regs do          { Bei "Remove Directory" zeigen DS:DX }
     p:= Ptr(DS,DX);     { auf den Namen (ASCIIZ-Format) }
   x:= 0;
   while p^[x] <> #0 do
   begin
     DirName[x+1] := p^[x];  { Kopie des ASCIIZ-Strings in die }
     Inc(x);                 { Pascal-Stringvariable DirName }
   end;
   DirName[0] := chr(x);     { Festlegen der Länge }
```

Die DOS-Funktion »Remove Directory« erwartet einen Zeiger auf den Namen des zu löschenden Directories im Registerpaar DS:DX. ASCIIZ-Strings enthalten kein Längenbyte und enden mit einem NUL-Zeichen - unser Ersatz für »Remove Directory« muß diese Angabe erst einmal in einen Pascal-String umwandeln.

```
   Write('Sind Sie sicher? (J/N) '); Readln;
   Write('Sind Sie wirklich sicher? (J/N) '); Readln;
   Writeln ('Und wenn ich nicht will, hä? Was dann?');
   for y:= 1 to 3 do
   begin
     x:= 0;
     while x < 300 do
     begin
       Sound(800+x); Delay(6);
       Sound(800-x); Delay(6);
       x := x + 20;
     end;
     NoSound; Delay(50);
   end;
   Delay(5000);
   Color := TextAttr;  { Originalfarbe speichern }
   TextAttr := Blink + Blue shl 4 + Red;
   Write('MÖRDER!':43); ClrEol;
   TextAttr := Color; Writeln;
```

Das könnte man natürlich noch bis in alle Ewigkeit fortsetzen...

```
    RmDir(DirName);

    with Regs do
    begin
      AX := IOResult;       { Ergebniscode }
      if AX <> 0
        then Flags := Flags OR $01       { Carry-Flag setzen }
        else Flags := Flags AND $FFFE;   { Carry-Flag löschen }
    end; { with Regs }
end;
```

Letztendlich tut das Programm aber, was man von ihm erwartet. Da *innerhalb* von *PasProc* der »echte« DOS-Vektor gesetzt ist, können wir *RmDir* hier problemlos verwenden. Der darauffolgende Aufruf von *IOResult* liefert den (von DOS erzeugten) Statuscode, den *PasProc* nun seinerseits an das Programm zurückliefert, das die DOS-Funktion $3A aufgerufen hat. (In den meisten Fällen wird es sich dabei um COMMAND.COM handeln). Da DOS bei der Funktion »Remove Directory« einen Fehler hauptsächlich über das Carry-Flag anzeigt, wird dieses Flag in *PasProc* anhand des Ergebnisses von *IOResult* gesetzt.

```
{$I INTCHECK.INC}    { Prüfung auf Kommandozeilen-Parameter OFF, Setzen
                       der diversen globalen Variablen usw. }
```

Die Datei INTCHECK.INC enthält die Funktion *IntCheck* (mit zwei Parametern) - Listing und Besprechung finden Sie im Abschnitt »Entfernbare TSR-Programme« auf Seite 256.

```
begin
  if not IntCheck(DOSINT,SPtr) then Halt;

  GetIntVec(DOSINT,SaveIntVec);  { Originalwert des zu setzenden Vektors }

  OrgIntOff := PWord(SaveIntVec).Off;
  OrgIntSeg := PWord(SaveIntVec).Seg;
  Prefix := PrefixSeg;
  CheckBreak := False;

  {$IFDEF VER50}
    SetIntVec($3F, SaveInt3F); { V 5.0: Overlay-Vektor zurück }
  {$ENDIF}
  SetIntVec($00, SaveInt00);  { beide Versionen: DIV0-Vektor zurück }
  SetIntVec($1B,SaveInt1B);       { ^Break-Vektor von CRT zurück }

  Writeln('Programm installiert');
  p := @DosOrg; p^:=  SaveIntVec;
  SetIntVec(DOSINT,@IntFrame);    { Installation der Routine }
  Keep(0);  { TSR, Exit-Code 0 }
end.
```

TPCED - Ein Kommandozeilen-Editor

TPCED stellt - neben TRACE287 - sozusagen die Krönung meiner Ideen zu TSR-Programmen dar. Für den Fall, daß Sie (wie ich bis vor wenigen Wochen) noch nie mit einem solchen Programm gearbeitet haben, erst einmal eine kurze Erläuterung, worum es sich dabei handelt:

COMMAND.COM bietet die folgenden Funktionen zur Korrektur von Tippfehlern, Wiederholung von Befehlen usw.: F3 bringt die zuletzt eingegebene Zeile wieder auf den Bildschirm, F2 mit einem danach eingegebenen Zeichen wiederholt diese Zeile bis zum ersten Vorkommen dieses Zeichens. Mit den Links- und Rechtspfeiltasten können einzelne Zeichen gelöscht bzw. wiederhergestellt werden - und das war's auch schon. Wer bei einem Befehl wie

```
copy \TP\PROGRAMS\FINAL\*.ASM \WORD\TP40\LISTS
```

irgendwo einen Schrägstrich oder einen Buchstaben vergißt, tippt die gesamte Zeile am besten neu ein - einen direkten Modus, über den zusätzliche Zeichen eingefügt werden könnten, kennt COMMAND.COM nicht. Ähnliches gilt, wenn man nach einem derartigen Befehl als nächstes `dir` eingibt und dann feststellt, daß auch noch alle .PAS-Dateien zu kopieren gewesen wären - mehr als die Wiederholung der jeweils letzten Kommandozeile ist nicht drin.

TPCED merkt sich die jeweils 30 zuletzt eingegebenen Kommandozeilen, kann zwischen »Einfügen/Überschreiben« umschalten, unterstützt HOME, END, wortweises Bewegen des Cursors, das Löschen einzelner Worte usw., wobei dieselben Tastenbefehle wie im Editor von Turbo Pascal verwendet werden.

Um etwas präziser zu werden:

Tabelle 4.1 - Tastenbefehle von TPCED

Taste	Wirkung
[↑]	bringt beim ersten Mal die letzte, beim zweiten Mal die als vorletztes eingegebene Kommandozeile auf den Bildschirm, d.h. bewegt sich rückwärts innerhalb des Puffers
[↓]	bewegt sich vorwärts innerhalb des Puffers
[Home]	setzt den Cursor auf den Anfang der momentanen Zeile
[End]	setzt den Cursor hinter das Ende der momentanen Zeile
[←]	bewegt den Cursor innerhalb der momentanen Zeile um eine Buchstabenposition nach links, ohne das Zeichen zu löschen
[→]	bewegt den Cursor innerhalb der momentanen Zeile um eine Buchstabenposition nach rechts
[Ctrl][A]	bewegt den Cursor innerhalb der momentanen Zeile um ein Wort nach links bzw. setzt ihn auf den Anfang des momentanen Wortes

Tabelle 4.1 - Tastenbefehle von TPCED (Fortsetzung)

Taste	Wirkung
[Ctrl][F]	bewegt den Cursor innerhalb der momentanen Zeile um ein Wort nach rechts
[Ctrl][T]	löscht das Wort rechts von der momentanen Position des Cursors
[Ctrl][Y]	löscht die gesamte Zeile
[Del]	löscht das Zeichen, auf dessen Position sich der Cursor momentan befindet
[Ins]	schaltet zwischen »Einfügen« und »Überschreiben« um
[Esc]	macht Änderungen innerhalb der momentanen Zeile wieder rückgängig
[F3]	ist ein Zugeständnis an alte Gewohnheiten - dieser Tastenbefehl bringt die jeweils letzte Eingabezeile wieder auf den Bildschirm und setzt den Cursor hinter ihr Ende, entspricht also der Befehlskombination [↑][End].

Nichts ist perfekt - es gibt einige Dinge, die das Programm *nicht* beherrscht bzw. unterstützt:

- den Tastenbefehl F2 von COMMAND.COM;
- den Abbruch einer Eingabezeile mit CONTROL-C oder CONTROL-BREAK;
- Kommandozeilen mit mehr als 79 Zeichen. Die Erweiterung auf einen »mehrzeiligen« Betrieb erhöht erstens den Bedarf an Speicherplatz recht kräftig, zweitens haben wir mit einem möglichen Rollen des Bildschirms zu kämpfen, wenn in der untersten Zeile des Bildschirms eine überlange Kommandozeile ausgegeben wird. Ich habe das Programm allerdings seit rund drei Wochen (über AUTOEXEC.BAT) im ständigen Betrieb und bin noch nie an diese Grenze gestoßen.
- Die Umschaltung der Druckerausgabe mit CONTROL-P. Das liegt daran, daß die Tastatur über die Funktion *ReadKey* des Units *Crt* gelesen wird (und DOS deshalb nichts davon mitbekommt). Die Beseitigung dieses Schönheitsfehlers ist unerwartet kompliziert und deshalb in einen eigenen Abschnitt verbannt worden: wir bleiben vorläufig bei der Standardversion von *ReadKey*.

TPCED ist wesentlich länger und aufwendiger als alle anderen in diesem Kapitel vorgestellten Programme - aus diesem Grund fangen wir mit einer »normalen« Programmversion an, um die später DOSCATCH.ASM, INTCHECK.INC und die restlichen Befehle zur Installation »herumgebaut« werden.

TPCED0.PAS - die »Trockenversion«

```
program TPCED0;   { Turbo Pascal Command Line Editor - 22/12/87 as }
uses Crt, Dos;

type
  EdResult = (ESC,CR,UP,DOWN,F3);    { ESC, RETURN, Pfeil auf, Pfeil ab, F3 }

const
  BufSize = 29;              { 30 Zeilen Puffer }
  LineLen = 70;              { Maximallänge einer Zeile }
  Init: Boolean = True;      { Flag: Puffer einmal durchlaufen }
  InsMode: Boolean = True;   { Insert/Overwrite }
  FirstLine: Integer = 0;    { Index in Lines: jeweils "unterste" Zeile }
  LastLine : Integer = 0;    { Index in Lines: jeweils letzte Zeile }

var
  Lines: array[0..BufSize] of string[LineLen];
  WLine: string[LineLen];
```

Die möglichen Ergebnisse der Bearbeitung einer Zeile über die Funktion *EditLine* sind hier als Aufzählungstyp (*EdResult*) deklariert. *BufSize* legt die Anzahl der Zeilen fest, die TPCED0 maximal speichert - weil wir wegen dem »Rückwärtsblättern« über ↑ mit einer Modulo-Funktion arbeiten müssen, der Compiler berechnete Konstanten erst seit der Version 5.0 akzeptiert und ich Ihnen einen Stacheldrahtverhau aus bedingten Compilerbefehlen ersparen wollte, hat *BufSize* einen um eins zu niedrigen Wert.

In *WLine* ist die jeweils aktuelle Zeile gespeichert. Der »gute Programmierstil« würde wohl verlangen, diesen String lokal zu *EditLine* zu deklarieren und als Funktionsergebnis zu übergeben - aber wer freut sich über ein schönes Listing, wenn der Prozessor jedesmal 256 Bytes umschaufelt (und dazu entsprechend viel Platz auf dem Stack braucht)?

```
procedure PasProc;
var
    Cx, Cy: Byte;          { Cursorposition: x,y }
    CurrentLine: Integer;  { Index: momentane Zeile in Lines }
    LastF3: Boolean;       { Flag: Abbruch mit F3 }
    GotLine: Boolean;      { Ende-Flag }
    Temp: Integer;

procedure ReprintWLine;    { Neuausgabe von WLine }
begin
  GotoXY(Cx,Cy); Write(WLine); ClrEol;
end;
```

Aus *PasProc* besteht praktisch das gesamte Programm - ähnlich wie bei *IntCheck* sind so viele Variablen wie möglich lokal (d.h. auf dem Stack) gespeichert.

ReprintWLine gibt die aktuelle Zeile vollständig neu aus und löscht danach den Rest der jeweiligen Bildschirmzeile. Diese Routine wird beim Wechsel zwischen Zeilen und nach dem Löschen ganzer Wörter aufgerufen.

```
function FindWordStart(Index,Direction: Integer): Integer;
const
  Alphas: set of Char = ['0'..'9','A'..'Z','a'..'z',
                         'ä','ö','ü','ß','Ä','Ö','Ü'];
var x,MinMax: Integer;
    XFlag: Boolean;   { für undurchsichtige Operationen...}
begin
  if Direction = 1 then   { Suche nach rechts, nächster Wortanfang }
    MinMax := Ord(WLine[0])+1 { Grenze: Zeilenlänge }
   else MinMax := 0; { Suche nach links, Grenze: Zeilenanfang }
  XFlag:= (Direction = -1);  { True für Suche nach links}

  x:= Index;  { momentane Position }
  if XFlag and (x > 1) then Dec(x); { für Links: Wortanfänge }
  while (x <> MinMax) and (XFlag xor (WLine[x] in Alphas)) do
    x := x + Direction;  { Rechts: Suche nach 1. Nicht-Alpha }
                         { Links: Suche nach 1. Alpha }
  if x <> MinMax then x:= x + Direction;
  while (x <> MinMax) and
    (XFlag xor (not (WLine[x] in Alphas))) do
    x := x + Direction;  { Rechts: Suche nach 1. Alpha }
                         { Links: Suche nach 1. Nicht-Alpha }
  if (XFlag and (x < MinMax)) or
   (not XFlag and (x > MinMax)) then x := MinMax;

  FindWordStart := x;
end;
```

FindWordStart liefert einen Index auf den Anfangsbuchstaben des nächsten Wortes zurück. Bei der Bewegung des Cursors nach links (mit CONTROL-A) müssen wir zwischen drei Möglichkeiten unterscheiden: 1) innerhalb eines Wortes -> Cursor auf den Wortanfang; 2) auf dem ersten Buchstaben eines Wortes -> Cursor zum Anfang des davorliegenden Wortes; 3) Cursor zwischen zwei Worten -> wie 2.

Für die Bewegung nach rechts mit CONTROL-F und das Löschen eines Wortes mit CONTROL-T ist die Endposition dagegen auf den nächsten Wortanfang festgelegt.

Die Trickserei mit der boolschen Variablen *XFlag* ist ein klassisches Beispiel für undurchsichtige Programmierung - wir haben in einem TSR-Programm aber keinen Platz für »schöne« Operationen*).

*) Außerdem mußte es einfach sein: Gregor hat in der Editor Toolbox (Version 4.0) für dieselbe Sache über zweieinhalb Seiten gebraucht - und zwar so gut wie unkommentiert.

```
function EditLine: EdResult;
var
  Index: Integer;
  Done: Boolean;
  Ch: Char;
  CharCode: Integer;   { für die Auswertung von Funktionstasten }
begin
  ReprintWLine;  { Ausgabe der Zeile }
  if LastF3 then
  begin                          { wenn der letzte Aufruf von }
    LastF3 := False;             { EditLine mit F3 abgebrochen }
    Index := Length(WLine)+1     { wurde: Cursor ans Zeilenende }
   end
    else Index := 1;
  Done:= False;
```

EditLine - die Funktion zur Bearbeitung der »aktuellen« Zeile - beginnt mit der Ausgabe dieser Zeile. Das Setzen des Cursors hinter das Zeilenende über F3 ist wie bereits angemerkt ein Zugeständnis an alte Gewohnheiten und paßt auch ansonsten nicht so recht in das Programm.

```
  repeat
    GotoXY(Cx+Index-1,Cy);   { Cursor setzen }
    Ch:= ReadKey;
    if Ch >= ' ' then
    begin
      if InsMode then
      begin
        Insert(Ch,WLine,Index);        { Einfügen des Zeichens }
        Write(Copy(WLine,Index,255)); { Ausgabe des Zeichens und aller }
        if Index < LineLen             { folgenden Zeichen }
          then Inc(Index);
       end
      else begin
        WLine[Index] := Ch;            { Überschreiben des Zeichens }
        Write(Ch);                     { Ausgabe des neuen Zeichens }
        if WLine[0] < Chr(Index) then WLine[0] := Chr(Index);
        if Index < LineLen then Inc(Index);
      end;
    end  { if Ch >= ' ' }
```

Ein »normales« Zeichen wird abhängig vom gesetzten Modus entweder mit *Insert* eingefügt (diese Routine begrenzt *WLine* automatisch auf die maximal zulässige Länge) oder direkt gesetzt. Da der Cursor bereits vor der Eingabe auf der richtigen Position im Bildschirm steht, können wir den erweiterten String bzw. das neue Zeichen ohne einen vorherigen Aufruf von *GotoXY* ausgeben.

Im Modus »Überschreiben« bleibt der Cursor beim Erreichen der Maximallänge *LineLen* auf dieser Position stehen - weitere Eingaben ersetzen in diesem Fall immer wieder das letzte Zeichen von *WLine*.

```
else
begin  { Ch ist kleiner als ' ' }
  if Ch = #0 then CharCode := (Ord(ReadKey)+1000)
     else CharCode := Ord(Ch);
  case CharCode of
       13:  begin  { RETURN }
              EditLine := CR; Done := True;
            end;
     1072: begin  { Aufwärtspfeil }
              EditLine := UP;
              Done := True;
            end;
     1080: begin { Abwärtspfeil }
              EditLine := DOWN;
              Done := True;
            end;
       27:  begin { ESC }
              EditLine := ESC;
              Done := True;
            end;
     1061: begin  { F3 - Verbinden mit dem Rest der vorherigen Eingabe }
              Delete(WLine,Index,255);  { Abschneiden }
              EditLine := F3;
              LastF3 := True;
              Done := True;
           end;
```

Wenn es sich bei der Eingabe um eine Funktionstaste handelt, dann wird hier einfach der Wert 1000 zum Scancode dazuaddiert. Auf diese Weise sparen wir uns eine weitere case-Anweisung und können für Funktionen wie CONTROL-G und DEL denselben Zweig verwenden.

(Die fünf Möglichkeiten, *EditLine* zu beenden, hätte man sicher noch etwas eleganter programmieren können. Tatsächlich belegt aber die Ausführung der ersten vier Zweige jeweils 6 Bytes - kürzer geht es mit einer Schleife auch nicht).

```
1082: InsMode := not InsMode;  { INS }
1075: if Index > 1 then Dec(Index); { Linkspfeiltaste }
1077: if Index <= Ord(WLine[0]) then Inc(Index); { Rechtspfeil }
1071: Index := 1;  { HOME }
1079: Index := Ord(WLine[0]) + 1; { END }
```

Die einfachen Steuerfunktionen: INS, Links- und Rechtspfeil, HOME und END.

```
          08:  if Index > 1 then { Backspace, ^H }
                 begin
                   Dec(Index);              { Zeichen links vom }
                   Delete(WLine,Index,1);   { Cursor löschen }
                   ReprintWLine;            { Neuausgabe }
                 end;
          07,
         1083: if WLine[0] > Chr(0) then { Ctrl-G, DEL }
                 begin
                   Delete(WLine,Index,1);   { Zeichen auf der Cursorpos. }
                   ReprintWLine;
                 end;
    {^A}   01: Index := 1 + FindWordStart(Index,-1);  { Word links }
    {^F}   06: Index := FindWordStart(Index,1);   { Wort rechts }
    {^T}   20: begin { Wort rechts löschen }
                 Delete(WLine,Index,FindWordStart(Index,1)-Index);
                 ReprintWLine;
               end;
    {^Y}   25: begin { Zeile komplett löschen }
                 WLine := ''; Index := 1;
                 ReprintWLine;
               end;
          end; { case }
      end; { else }
    until Done;
  end;
```

Die Funktionen zum Löschen von Zeichen, eines ganzen Wortes, der gesamten Zeile und der wortweisen Bewegung des Cursors.

```
  begin   { PasProc }
    WLine := ''; Cx:= WhereX; Cy:= WhereY;
    CurrentLine := LastLine; GotLine := False;
    Lines[LastLine] := ''; LastF3 := False;
```

Das Hauptprogramm von *PasProc*: Arbeitszeile und zukünftige »Ziel-Zeile« werden gelöscht, *Cx* und *Cy* halten die momentane Position des Cursors fest. Welchen Anfangswert *Cx* bekommt, hängt von der Länge des momentanen Prompts ab.

```
    repeat
      case EditLine of
        CR: begin   { Abschluß der Eingabezeile }
              Lines[LastLine] := WLine;   { Eintragen }
              LastLine := (LastLine + 1) mod (BufSize + 1);
              if LastLine = 0 then Init := False;  { Puffer ist jetzt voll }
              if not Init then
                FirstLine := (FirstLine + 1) mod (BufSize + 1);
              GotLine := True;
            end;
```

```
      ESC: WLine := Lines[CurrentLine]; { Neustart }
       F3: if LastLine <> FirstLine then { Verbinden mit dem Rest der }
           begin                         { vorherigen Eingabe }
             Temp := LastLine-1;
             if Temp < 0 then Temp := BufSize;
             WLine := WLine + Copy(Lines[Temp],Length(WLine)+1,255);
           end;
       UP: begin { Abbruch mit Aufwärtspfeiltaste }
             if CurrentLine <> FirstLine then
             begin
               Dec(CurrentLine);  { mod funktioniert hier nicht! }
               if CurrentLine < 0 then CurrentLine := BufSize;
             end;
             WLine := Lines[CurrentLine];  { auch bei CurrentLine = FirstLine }
           end;
     DOWN: begin { Abbruch mit Abwärtspfeiltaste }
             if CurrentLine <> LastLine then
               CurrentLine := (CurrentLine + 1) mod (BufSize + 1);
             WLine := Lines[CurrentLine]; { auch bei CurrentLine = LastLine }
           end;
      end; { case }
    until GotLine;
  end; { PasProc }
```

Das Flag *Init* wird beim Start des Programms auf *True* gesetzt und erst dann gelöscht, wenn der Puffer von TPCED komplett gefüllt ist - ab diesem Zeitpunkt muß auch der Index zur »ersten Zeile« (*FirstLine*) bei jeder Speicherung erhöht werden. Bei einer Programmierung ohne diesen Anfangsindex bekäme der Benutzer Undefiniertes zu sehen, wenn er innerhalb der ersten dreißig Eingaben zu oft auf die Aufwärtspfeiltaste drückt.

Der Abbruch von *EditLine* mit F3 erzeugt dasselbe Ergebnis wie innerhalb von COMMAND.COM: an die momentane Arbeitszeile wird der Rest der vorherigen Eingabe ab der Cursorposition angehängt.

Die Speicherung einer mit RETURN abgeschlossenen Zeile geschieht unabhängig von *CurrentLine* in dem Element von *Lines*, auf das der Index *LastLine* zeigt. Diese Zeile steht deshalb immer als unterste (oder oberste - wie man es sehen will) in *Lines* und wird durch die erste Eingabe von ↑ beim nächsten Aufruf von *PasProc* wieder auf den Bildschirm geholt.

```
    { *** "Hauptprogramm", muß mit ^Break abgebrochen werden *** }
  begin
    Init := True;  { Puffer ist noch nicht voll }
    repeat
      PasProc;
      Writeln;
    until False;
  end.
```

Das »Hauptprogramm« von TPCED0 ist wirklich denkbar kurz ausgefallen - es ruft *PasProc* solange immer wieder auf, bis es mit CONTROL-BREAK abgebrochen wird. Der jeweils nachfolgende Aufruf von *Writeln* ist notwendig, weil *PasProc* selbst keinen Zeilenvorschub ausgibt.

Ein **Hinweis zur Version 5.0:** Wenn Sie TCED0 von der integrierten Entwicklungsumgebung aus starten, sind zwei aufeinanderfolgende Eingaben von CONTROL-BREAK zum Abbruch erforderlich.

TPCED als speicherresidentes Programm

Das folgende Listing beschränkt sich auf die Erweiterungen, die aus TPCED0 ein TSR-Programm machen. Soweit notwendig, sind unterschiedliche Programmteile zusätzlich fett gesetzt.

TPCED ersetzt die DOS-Funktion $0A (»Buffered Keyboard Input«) und besteht aus vier Teilen:

- CATCH22.PAS mit FUNCNO = $0A (ohne *PasProc*);
- INTCHECK.INC (wie in CATCH22);
- DOSCATCH.ASM (wie in CATCH22);
- den Deklarationen und der Prozedur *PasProc* aus TPCED0.PAS mit einer zusätzlichen Routine (*CopyWLine*), die die Eingabezeile an das aufrufende Programm übergibt. Diese Routine ist der Einfachheit halber global deklariert und wird am Ende von *PasProc* aufgerufen.

```
program TPCED_Resident;  { CED-Clone in Turbo Pascal - 22/12/87 as }

{$M 2000,0,0}              { *** !!! *** }
{$S-,R-,I-}
uses Crt,Dos;    { wg. GotoXY bzw. Registers, SetIntVec usw. }
type
  PWord = record   { Typ-Umwandlung für Zeigerwerte }
     Off: Word;       { Offset }
     Seg: Word;       { Segment }
   end;

  ThreeIDs = array[0..2] of Byte; { Typ-Umwandlung für IDs }

const
  IDs: ThreeIDs = (22,12,87);   { ID des Programms }
  OrgIntOff: Word = 0;          { Hier wird der Original- }
  OrgIntSeg: Word = 0;          { Interrupt-Vektor gesetzt }
  Prefix: Word = 0;             { und hier das Programmsegment-Präfix }

  FUNCNO:Byte = $0A;  { Diese Funktion wird durch PASPROC ersetzt }
  DOSINT = $21;  { Dieser Interrupt wird ersetzt }
```

```
var
  TurboSS, TurboSP: Word;  { SS:SP von Turbo }
  SaveIntVec: Pointer;     { zum Speichern des Originalvektors }
  p: ^Pointer;

  Regs: Registers;         { Übergabe der Registerwerte }
  CallCS, CallIP: Word;    { CS:IP des aufrufenden Programms }
  SSSave, SPSave: Word;    { SS:SP des aufrufenden Programms }

procedure IntFrame; external;
procedure DosOrg; external;     { muß auf den DOS-Originalvektor gesetzt
                                  werden }
{$L DOSCATCH}
```

Dieser Deklarationsteil kann praktisch unverändert aus CATCH22 übernommen werden - lediglich die beiden fettgedruckten Zeilen sind entsprechend zu ändern.

```
type
  EdResult = (ESC,CR,UP,DOWN,F3);    { ESC, RETURN, Pfeil auf, Pfeil ab, F3 }

const
  BufSize = 29;              { 30 Zeilen Puffer }
  LineLen = 70;              { Maximallänge einer Zeile }
  Init: Boolean = True;      { Flag: Puffer einmal durchlaufen }
  InsMode: Boolean = True;   { Insert/Overwrite }
  FirstLine: Integer = 0;    { Index in Lines: jeweils "unterste" Zeile }
  LastLine : Integer = 0;    { Index in Lines: jeweils letzte Zeile }

var
  Lines: array[0..BufSize] of string[LineLen];
  WLine: string[LineLen];
```

Die Deklaration der globalen Variablen aus TPCED0 ist unverändert.

```
    { ------- neue Prozedur --------- }
procedure CopyWLine;  { kopiert die Eingabezeile (WLine) in den }
var                   { Arbeitspuffer des aufrufenden Programms }
  p: ^string;
  Max,x: Integer;
begin
  with Regs do
    p := Ptr(DS,DX);  { DS:DX des aufrufenden Programms = ^Arbeitspuffer }
  Max := Ord(p^[0]);  { Byte 0: Maximalgröße }
  x := Ord(WLine[0]); { Stringlänge }
  if x > Max-3 then x := Max -3;
  Inc(PWord(p).Off);  { p auf Byte 1 des Puffers }
  p^ := Copy(WLine,1,x);
  p^[x+1] := #13;  { CR anhängen }
end;
```

Ein Programm, das die DOS-Funktion $0A verwendet, muß das Registerpaar DS:DX auf den Speicherbereich setzen, in dem der String abgelegt werden soll. Das erste Byte dieses Speicherbereichs wird vom Programm auf die Maximallänge des Strings gesetzt, in das zweite setzt DOS die Anzahl der tatsächlich gelesenen Zeichen, ab dem dritten Byte folgen die Zeichen selbst. DOS hängt an diesen String zusätzlich das abschließende RETURN an, ohne es im 2. Byte (d.h. der Stringlänge) mitzuzählen. *CopyWLine* simuliert diese Form der Übergabe exakt (inklusive der Speicherung des nicht mitgezählten RETURN).

```
procedure PasProc;
var
    Cx, Cy: Byte;          { Cursorposition }
   .....

   .....
   DOWN: begin { Abbruch mit Abwärtspfeiltaste }
           if CurrentLine <> LastLine then
             CurrentLine := (CurrentLine + 1) mod (BufSize + 1);
           WLine := Lines[CurrentLine]; { auch bei CurrentLine = LastLine }
         end;
    end; { case }
  until GotLine;

   CopyWLine;     { kopiert die neue Eingabezeile -> DS:DX }
end; { PasProc }
```

Die einzige Veränderung innerhalb von *PasProc* ist der Aufruf von *CopyWLine* vor dem Ende der Prozedur.

```
{$I INTCHECK.INC}    { Prüfung auf Kommandozeilen-Parameter OFF, Setzen
                       der diversen globalen Variablen usw. }

{ ------------------------------------------------------------ }

begin  { *** ACHTUNG: Die folgende Zeile *muß* mit aufgenommen werden -
         ansonsten funktioniert das Programm garantiert nicht! }
  Writeln('TPCED - Turbo Pascal-Kommandozeilen-Editor (c) as 12/87');

  if not IntCheck(DOSINT,SPtr) then Halt;

  GetIntVec(DOSINT,SaveIntVec);  { Originalwert des zu setzenden Vektors }

  OrgIntOff := PWord(SaveIntVec).Off;  { Eckwerte für die spätere }
  OrgIntSeg := PWord(SaveIntVec).Seg;  { Entfernung des Programms }
  Prefix := PrefixSeg;

  Init := True;                   { Puffer ist nicht voll }
```

```
  {$IFDEF VER50}
    SetIntVec($3F, SaveInt3F); { V 5.0: Overlay-Vektor zurück }
  {$ENDIF}
  SetIntVec($00, SaveInt00);  { beide Versionen: DIV0-Vektor zurück }
  SetIntVec($1B,SaveInt1B);       { ^Break-Vektor von Crt zurück }

  CheckBreak:= False;   { *** ! *** }
  DirectVideo := False;

  Writeln('Programm installiert');

{ *** Achtung: Die Installation muß der letzte Befehl vor KEEP sein -
  ansonsten würde sich das Programm selbst unterbrechen! }
  p := @DosOrg; p^:=  SaveIntVec;

  SetIntVec(DOSINT,@IntFrame);    { Installation der Routine }
  Keep(0);  { TSR, Exit-Code 0 }
end.
```

Das Hauptprogramm von TPCED. Die Include-Datei INTCHECK.INC bzw. die darin enthaltene Funktion *IntCheck* ist dieselbe wie in CATCH22, TRACE86R etc., Listing und Besprechung finden Sie im Abschnitt »Hardware-Interrupts« auf Seite 256. Was das »Achtung« vor dem Aufruf von *IntCheck* betrifft: hier gilt dasselbe wie bei TRACE287.

Das Hauptprogramm von TPCED unterscheidet sich von den restlichen Beispielen dieses Kapitels in zwei Punkten:

- Dem Setzen des Flags *Init* für die Anfangs- und Endindexe innerhalb von *PasProc*. Falls *Init* undefiniert ist, erhalten Sie bei den ersten Eingaben nach dem Start von TPCED schlimmstenfalls einige wunderliche Zeichen auf dem Bildschirm.
- Dem Zurücksetzen von *CheckBreak*, *DirectVideo* und des *Break*-Vektors von *Crt*. Für *DirectVideo* gilt dasselbe wie bei TRACE287 - wenn diese Variable auf *True* gesetzt ist, schreibt *Write* direkt in den Bildspeicher und Sie bekommen nichts mehr zu sehen, wenn Sie zwischenzeitlich in einen Grafikmodus umgeschaltet haben.

Das Zurücksetzen von *CheckBreak* und des *Break*-Vektors ist bei TPCED **lebensnotwendig**. Wenn diese Zuweisungen nicht ausgeführt werden, versucht *Crt* bei der Eingabe von CONTROL-BREAK, die Ausführung von TPCED über den DOS-Aufruf $4C (»Terminate«) abzubrechen - eine Operation, die einem TSR-Programm (und dem System) nicht sonderlich gut bekommt. Da der Interruptvektor $21 während der Ausführung von *PasProc* wieder auf DOS gesetzt ist, kommt es nicht sofort zum Absturz; durch dieses Hauruckverfahren scheinen aber noch einige andere Dinge durcheinanderzukommen, deren Natur nicht bis ins letzte erforscht ist.

Eine neue Version von *ReadKey*

Wie zu Anfang der Besprechung von TPCED angedeutet, benötigen wir zur Unterstützung des Tastenbefehls CONTROL-P eine eigene Version von *ReadKey*, weil DOS einen Funktionsaufruf wie »Echo to Printer on/off« nicht kennt.

Tatsächlich funktioniert die Umschaltung des »Drucker-Echos« nur dann, wenn der Befehl CONTROL-P während einer Ein- oder Ausgabeoperation von DOS gegeben wird:

- Bei Ausgaben prüft DOS nach jedem Zeilenvorschub, ob in der Zwischenzeit ein CONTROL-P eingegeben wurde, und schaltet gegebenenfalls die Druckerausgabe um. Wenn Sie beispielsweise eine Textdatei mit `type` ausgeben lassen und während des Bildschirmrollens ein CONTROL-P eingeben, funktioniert die Umschaltung auch mit der bisher vorgestellten Version von TPCED.
- Bei Eingabeanforderungen verarbeitet DOS ein CONTROL-P direkt. Das Zeichen wird nicht an das Programm weitergegeben, das die Eingabe angefordert hat - stattdessen wartet DOS gleich auf das nächste Zeichen.

Aus dem (an sich vernünftigen) Verhalten bei Eingaben ergibt sich ein Problem, zu dessen Lösung Erstaunliches in Bewegung gesetzt werden muß. Die diversen Hürden sind:

1. Über den BIOS-Interrupt $16, Subfunktion 1 können wir prüfen, welches Zeichen eingegeben wurde, ohne es dabei »abzuholen«. Es besteht also die Möglichkeit, dem Betriebssystem ein im voraus erkanntes CONTROL-P durch einen entsprechenden Aufruf zu lesen zu geben. Da dieses CONTROL-P aber direkt verarbeitet wird, wartet DOS danach auf eine zweite Eingabe.
2. Die diversen DOS-Funktionen zum Lesen eines einzelnen Zeichens reagieren entweder auf CONTROL-P und CONTROL-C bzw. CONTROL-BREAK (Funktionen 1 und 8) oder auf überhaupt keines dieser Steuerzeichen (Funktionen 6 und 7).
3. CONTROL-C und CONTROL-BREAK werden nirgendwo so häufig benutzt wie auf der Kommandoebene von DOS, müssen also von TPCED in vernünftiger Form bearbeitet bzw. abgefangen werden.
4. Ein globales Flag innerhalb von DOS, das über CONTROL-P umgeschaltet wird, ist zwar vorhanden, seine Adresse unterscheidet sich aber von Version zu Version. Die Möglichkeit, dieses Flag einfach direkt zu setzen, scheidet deshalb aus.

Etwas kürzer: a) Eine andere Methode als das »Lesenlassen« eines CONTROL-P durch DOS zur Umschaltung des Druckers ist mir nicht eingefallen; und b) wenn DOS nach diesem CONTROL-P als zweites Zeichen ein CONTROL-C bzw. CONTROL-BREAK liest, wird TPCED aus dem Speicher hinauskatapultiert bzw. das System stürzt ab.

Soviel zur Einleitung des nun folgenden, wahrhaft exotischen Lösungsvorschlags, der wie alle genialen Ideen wirklich simpel ist: Wenn das nächste Zeichen ein CONTROL-P ist, dann gibt die neue Version von *ReadKey* dieses Zeichen DOS zu lesen und sorgt einfach dafür, daß das *übernächste* Zeichen kein CONTROL-C ist. Punkt.

Wie das geht? Indem *ReadKey* nötigenfalls eine zweite (und »harmlose«) Eingabe im Tastaturpuffer einträgt. Nach der Erkennung eines CONTROL-P und vor der Übergabe an DOS müssen wir dazu zwischen vier möglichen Situationen unterscheiden:

1. Der Tastaturpuffer enthält momentan nur dieses eine Zeichen. *ReadKey* schreibt ein zweites Zeichen hinein und ruft DOS auf. DOS liest das CONTROL-P, verarbeitet es intern, liest das zweite Zeichen und liefert es an *ReadKey* zurück. Welches Zeichen wir in den Tastaturpuffer hineingeschrieben haben, spielt keine Rolle - *ReadKey* ignoriert dieses »Ergebnis« des DOS-Aufrufs sowieso und liefert CONTROL-P als Funktionsergebnis an *PasProc* bzw. *EditLine* zurück.

2. Der Tastaturpuffer enthält direkt nach dem ersten CONTROL-P ein zweites. Da sich diese beiden Befehle gegenseitig aufheben, werden sie von *ReadKey* (mit zwei Aufrufen von *Crt.ReadKey*) aus dem Puffer entfernt, danach beginnt *ReadKey* von vorne, d.h. wartet auf ein »lesbares« Zeichen.

3. Der Tastaturpuffer enthält direkt nach dem CONTROL-P ein CONTROL-C. *ReadKey* überschreibt dieses Zeichen mit etwas Unverfänglichem, läßt DOS das CONTROL-P sowie das nachfolgende Zeichen lesen und ignoriert das Ergebnis. Das Funktionsergebnis für diesen Aufruf von *ReadKey* ist ein CONTROL-P, das Ergebnis des *nächsten* Aufrufs ein CONTROL-C.

4. Der Tastaturpuffer enthält direkt nach dem CONTROL-P ein weiteres (und »harmloses«) Zeichen. *ReadKey* läßt DOS das CONTROL-P zusammen mit diesem zweiten Zeichen lesen. Das Funktionsergebnis für diesen Aufruf von *ReadKey* ist ein CONTROL-P, das Ergebnis des *nächsten* Aufrufs das zweite gelesene Zeichen.

Zur Organisation des Tastaturpuffers

Nach dieser erschöpfenden Betrachtung der Möglichkeiten und Verwicklungen bleibt nur noch die Frage zu klären, wie man dem Tastaturpuffer zusätzliche Zeichen hinzufügt bzw. einzelne Zeichen darin überschreibt.

Nach Peter Nortons *Programmer's Guide* verwendet der BIOS-Interrupt $16 unter anderem zwei *Word*-Variablen und einen Bereich von 32 Bytes im Segment $0040, deren Zweck und Adressen auf der nächsten Seite wiedergegeben sind:

Tabelle 4.2 - Tastaturpuffer und Indexvariablen

Adresse	Beschreibung
$40:$1A	*KbHead* - Index auf die Adresse, auf der das nächste zu lesende Zeichen im Tastaturpuffer steht, d.h. den momentanen Kopf der Warteschlange.
$40:$1C	*KbTail* - Index auf das Ende der Warteschlange.
$40:$1E ..... $40:$3C	Der Tastaturpuffer selbst: 16 Einträge mit jeweils 2 Bytes. Der ASCII-Code einer Eingabe ist jeweils im niederwertigen Byte, der Scancode im höherwertigen Byte gespeichert. Funktionstasten belegen hier ebenfalls nur 2 Bytes (und nicht 4).

Die beiden Zeiger (deren Namen eine freie Erfindung sind) haben etwas merkwürdige Wertebereiche: Sie enthalten jeweils eine vollständige Offset-Adresse innerhalb des Segments $0040, können also Werte im Bereich von $001E..$003C annehmen. Beide Zeiger bewegen sich im Kreis: *KbHead* wird durch jede Eingabe um zwei erhöht, *KbTail* durch jede Leseaktion. Wird der Wert $3C bei der Erhöhung eines Zeigers überschritten, dann bekommt dieser Zeiger wieder den Wert $1E zugewiesen.

Einige Beispiele dazu: Wenn *KbHead* den Wert $2A enthält und *KbTail* den Wert $2E, dann befinden sich zwei Zeichen im Tastaturpuffer, das erste Zeichen ist auf der Adresse $0040:$2A/$2B gespeichert, das zweite auf der Adresse $0040:$2C/$2D. Wenn *KbHead* den Wert $3C enthält und *KbTail* den Wert $2A, dann befindet sich ein Zeichen auf der Adresse $0040:$3C/$3E im Puffer.

Wenn *KbHead* denselben Wert wie *KbTail* enthält, wird der Tastaturpuffer als leer betrachtet; bei *KbHead* = *KBTail* - 2 ist der Puffer voll.

Dieser Aufbau bedingt eine Reihe zusätzlicher Rechenschritte bei direkten Zugriffen durch ein Pascal-Programm - erfreulicherweise scheinen hier aber keine weiteren Flags etc. mit im Spiel zu sein, d.h. das Einfügen eines zusätzlichen Zeichens beschränkt sich auf den direkten Eintrag und die entsprechende Erhöhung von *KbTail.*

Der Interrupt $1B

ist leider auch noch zu berücksichtigen - er wird (wie Print Screen) direkt von der Tastatur ausgelöst und führt im Normalfall zu einer Routine von DOS, die in der Version 3.2 so aussieht:

```
0070:0C63 2E            CS:
0070:0C64 C606D00B03    MOV     BYTE PTR [0BD0],03
0070:0C69 CF            IRET
```

Hier setzt DOS den ASCII-Code des nächsten über einen DOS-Aufruf zu lesenden Zeichens - nämlich CONTROL-C. Da der Interruptvektor $1B auch noch von einer ganzen Reihe anderer Programme gesetzt wird (nicht zuletzt durch jedes Turbo

Pascal-Programm, das das Unit *Crt* verwendet), ist ein völliges »Verbiegen« sicher keine gute Idee - außerdem sollte es die neue Version von *ReadKey* dem aufrufenden Programm erlauben, auch auf CONTROL-BREAK in sinnvoller Weise zu reagieren.

Jedesmal, wenn TPCED aktiv wird, setzen wir diesen Vektor deshalb auf eine als **interrupt** deklarierte Routine, die einen ähnlichen Umfang wie ihr Gegenstück in DOS hat:

```
procedure DummyBreak;  interrupt;
begin
  BreakTyped := True;
end;
```

In *ReadKey* wird bei jedem Aufruf zuerst geprüft, ob *BreakTyped* gesetzt wurde. Wenn ja, dann setzt *ReadKey* dieses Flag zurück und liefert CONTROL-C als Funktionsergebnis.

TPCED mit Erweiterungen

Bevor wir nun endlich zum Listing von *ReadKey* und dem Einbau in TPCED kommen, noch ein allerletztes Problem: Die Aufrufe von *Writeln* innerhalb von TPCED schreiben je nach Stand von *DirectVideo* entweder direkt in den Bildspeicher oder verwenden einen BIOS-Aufruf dazu: Nach einer Aktivierung des Drukkers mit CONTROL-P bekommen wir zwar die Ergebnisse zu sehen, nicht aber die Kommandozeilen selbst.

Im großen und ganzen sollte das auch so bleiben - schließlich sind ein halbes Dutzend »durchgeblätterte« Kommandozeilen auf dem Drucker (oder in einer mit PRTFILE erzeugten Datei) unerwünscht. Nur für die endgültige (d.h. mit RETURN abgeschlossene) Kommandozeile folgt deshalb der Aufruf einer neuen Prozedur namens *DOSPrintWLine*, die diese Zeile zeichenweise über einen DOS-Aufruf noch einmal ausgibt. Vor dem Aufruf von *DOSPrintWLine* wird der Cursor noch einmal auf den Anfang der Kommandozeile gesetzt - auf dem Bildschirm bleibt die Aktion unsichtbar, weil sich die Zeile selbst überschreibt.

```
procedure CopyWLine;
var
  p: ^string;
  Max,x: Integer;
begin
 .....
end;

 { ------- Beginn des neuen Teils ---------------- }

const
  BreakTyped: Boolean = False;       { wird durch ^Break gesetzt }
  PrintActive: Boolean = False;      { wird durch ^P negiert }
```

Die Deklaration dieser beiden Flags als typisierte Konstanten erspart uns die ansonsten notwendige Initialisierung (und damit die Modifikation weiterer Programmteile).

```
procedure DOSPrintWLine;
{ Gibt WLine bei aktiviertem Drucker über den Standardweg von DOS aus }
var
  x: Integer;
  Ch: Char;
begin
  for x:= 1 to Length(WLine) do
  begin
    Ch:= WLine[x];
    inline($B4/$02/        { MOV AH,2       ; "Display Output" }
           $8A/$56/<Ch/    { MOV DL,Ch[BP]  ; ein Zeichen von WLine }
           $CD/$21);       { INT $21        ; DOS-Aufruf }
   end;
end;
```

DOSPrintWLine wird nur dann am Ende von *PasProc* aufgerufen, wenn *PrintActive* auf *True* gesetzt ist.

```
procedure DummyBreak; interrupt;
begin
  BreakTyped := True;
end;

function ReadKey: Char;
const
  BUFSTART = $1E;       { Tastaturpuffer-Start: $40:$1E }
  BUFEND   = $3C;       { letzter Eintrag im Puffer: $40:$3C }
  CharBuf: Char = #0;   { Zwischenspeicherung bei "Doppeltastendrücken" }
  CBufSet: Boolean = False;  { Flag: "CharBuf ist gesetzt" }
var
    KbHead: Word absolute $40:$1A;   { Index zum Beginn der Warteschlange }
    KbTail: Word absolute $40:$1C;   { Index aufs Ende }
    KeyP : ^Word;                    { Zeiger auf ein Zeichen }
    NextChar: Word;                  { Index zum *nächsten* Zeichen }
    GotChar: Boolean;
    Ch: Char;
begin
  repeat
    if BreakTyped then           { für den Fall, daß uns hier jemand }
    begin                        { mit ^Break dazwischenfunkt }
      ReadKey := ^C;
      BreakTyped := False;
      Exit;
    end;
```

```
if CBufSet then
begin                          { für ^P^C oder ^P<normales Zeichen> }
  ReadKey := CharBuf;          { wird beim ersten Aufruf von ReadKey }
  CBufSet := False;            { das ^P zurückgeliefert und das 2. }
  Exit;                        { Zeichen in CharBuf zwischengespeichert }
end;
```

Wenn *BreakTyped* oder *CBufSet* gesetzt sind, dann bricht *ReadKey* sofort mit dem entsprechenden Ergebnis ab. Die Reihenfolge der Prüfungen entspricht der von DOS - CONTROL-BREAK wird vor allen anderen Eingaben bearbeitet.

```
GotChar := True;      { Annahme: wir bekommen ein Zeichen zu lesen }
repeat until KeyPressed;  { Warten auf einen Tastendruck }
KeyP := Ptr($40,KbHead);  { KeyP auf dieses Zeichen im Puffer setzen }
if KeyP^ <> $1910 then    { -> dieses Zeichen ist kein Ctrl-P }
  Ch := Crt.ReadKey
```

Über den Aufruf von *KeyPressed* wird sichergestellt, daß es überhaupt etwas zu lesen gibt. Wenn schließlich ein Zeichen eingegeben wurde und dieses Zeichen kein CONTROL-P ist (Scancode $19, ASCII-Code 16 = $10), dann wird es über die Standardfunktion *ReadKey* des Units *Crt* gelesen.

```
else
begin
  NextChar := KbHead + 2;   { Index zum *nächsten* Zeichen im Puffer }
  if NextChar > BUFEND then { .. die Adressierung läuft hier im Kreis }
    NextChar := BUFSTART;
  KeyP := Ptr($40,NextChar); { KeyP auf das nächste Zeichen }
  if KbTail = NextChar then
  begin                     { Möglichkeit 1: keine weiteren Zeichen }
    KeyP^ := $1E41;         { Unterschieben eines 'A' mit Erhöhung }
    Inc(KbTail,2);          { von KbTail }
    if KbTail > BUFEND then
      KbTail := BUFSTART;
    Ch := ^P;               { Ergebnis: Ctrl-P }
  end
  else if KeyP^ = $1910 then
  begin                     { Möglichkeit 2: es folgt ein 2. Ctrl-P }
    Ch := Crt.ReadKey;      { diese beiden Ctrl-Ps heben sich auf }
    Ch := Crt.ReadKey;      { und werden deshalb ignoriert }
    GotChar := False;       { danach Wiederholung der Schleife }
  end
  else if KeyP^ = $2E03 then
  begin                     { Möglichkeit 3: es folgt ein Ctrl-C }
    KeyP^ := $1E41;         { Verwandlung in ein "harmloses" 'A' }
    Ch := ^P;               { Ergebnis 1. Aufruf von ReadKey: Ctrl-P }
    CharBuf := ^C;
    CBufSet := True;        { Ergebnis 2. Aufruf von ReadKey: Ctrl-C }
  end
```

```
      else
      begin                    { -> es folgt ein normales Zeichen, }
        CharBuf:= Chr(Lo(KeyP^)); { das hier der Einfachheit halber gleich }
        CBufSet := True;       { gelesen und beim 2. Aufruf geliefert wird }
        Ch := ^P;              { Ergebnis 1. Aufruf von ReadKey: Ctrl-P }
      end;

     if GotChar then
       inline($B4/$08/       { MOV AH,08  ; "Keyboard Input w/o Echo" }
              $CD/$21);      { INT $21    ; DOS-Aufruf }
                { das Ergebnis ist bereits bekannt und wird ignoriert }
    end;  { für 1.Zeichen = Ctrl-P }
  until GotChar;
  ReadKey := Ch;
end;
```

Die Bearbeitung der vier zuvor besprochenen Möglichkeiten. Nur im zweiten Fall (zweimal CONTROL-P hintereinander) wird die Schleife wiederholt - ein DOS-Aufruf findet dabei nicht statt, weil wir diese beiden Zeichen durch Aufrufe von *Crt.ReadKey* bereits gelesen haben.

```
procedure PasProc;
var
    Cx, Cy: Byte;          { Cursorposition }
    CurrentLine: Integer;  { Index: momentane Zeile in Lines }
    GotLine: Boolean;      { Ende-Flag }
    LastF3: Boolean;       { Flag: Abbruch mit F3 }
    Temp: Integer;
    SaveInt1B: Pointer;    { für den ^Break-Vektor }

procedure ReprintWLine;    { Neuausgabe von WLine }
 .....
```

Die neue Variable *SaveInt1B* ist lokal zu *PasProc* deklariert und hat deshalb mit der gleichnamigen Variablen des Units *Crt* nichts zu tun. In ihr wird der originale Wert des *Break*-Vektors bei jeder Aktivierung von *PasProc* zwischengespeichert.

```
function EditLine: EdResult;
var
  Index: Integer;
  Done: Boolean;
  Ch: Char;
begin
  ...

   {^Y}   25: begin { Zeile komplett löschen }
                WLine := ''; Index := 1;
                ReprintWLine;
              end;
```

```
    {^P}  16: PrintActive := not PrintActive;  { Umschaltung mit Ctrl-P }
          end; { case }
      end; { else }
    until Done;
  end;
```

Die case-Anweisung in *EditLine* ist um einen Zweig für CONTROL-P erweitert worden.

```
  begin  { PasProc bzw. EditLine }
    GetIntVec($1B,SaveInt1B);
    SetIntVec($1B,@DummyBreak);
    WLine := ''; Cx:= WhereX; Cy:= WhereY;
    CurrentLine := LastLine; GotLine := False;
   ......

     CopyWLine;     { kopiert die neue Eingabezeile -> DS:DX }

      if PrintActive then { Drucker aktiv? }
      begin               { Ausgabe von WLine über DOS }
        GotoXY(Cx,Cy);
        DOSPrintWLine;
      end;

      SetIntVec($1B,SaveInt1B);
  end; { PasProc }
```

Zu Anfang des Hauptprogramms von *PasProc* wird der *Break*-Vektor in *SaveInt1B* gespeichert und danach für die Dauer der Prozedurausführung auf *DummyBreak* gesetzt.

Am Ende von *PasProc* steht die Prüfung von *PrintActive* und der davon abhängige Aufruf von *DOSPrintWline*, bevor der *Break*-Vektor wieder seinen ursprünglichen Wert bekommt. Die Reihenfolge dieser beiden Schritte ist kritisch: Wenn der *Break*-Vektor vor dem Aufruf von *DOSPrintWLine* zurückgesetzt würde, ließe sich das Programm mit CONTROL-BREAK an dieser Stelle (mit den bereits öfter erwähnten Folgen) unterbrechen.

Kapitel 5
Grafik

In diesem Kapitel geht es zwar auch um die Analyse von Strukturen, Abläufen und Prinzipien - Vergleiche im Stile von »aus der Programmzeile xxx macht der Compiler« werden Sie allerdings nur wenige finden. Der Grund: wir haben hier einen lebenden Beweis für die universelle Gültigkeit der Formel

$$e = mc^2$$

- wobei *m* für »Möglichkeiten«, *c* für eine Confusionsconstante und *e* für die sich ergebende Entropie steht. Die Möglichkeiten des Units *Graph* sind so ausgeprägt, daß der Wert von *c* getrost mit $3*10^9$ angenommen werden darf.

Etwas unwissenschaftlicher ausgedrückt: Weil *Graph* so programmiert wurde, daß man nicht nur die Treiber und Zeichensätze beliebig wechseln kann, sondern auch die Zielsprache*), springt der Prozessor innerhalb dieses Units so hin und her, daß einem bereits beim Zusehen schwindlig wird.

Deshalb geht es in den nächsten Abschnitten nur in den wenigsten Fällen um den tatsächlichen Code von *Graph* - wir konzentrieren uns stattdessen auf:

- eine allgemeine Analyse der Struktur;
- die Erweiterung der (statischen) Grafiktreiber für Grafik-Karten, die mehr als die üblichen 640 * 350 Pixel darstellen können;
- den Aufbau der Zeichensätze und ihre Erweiterungsmöglichkeiten;
- das Programm CE - einen Editor für grafische Zeichensätze;
- und das Laden und Speichern von Grafikbildern (unabhängig vom verwendeten Video-Adapter und Grafikmodus).

Zur Struktur von *Graph*

An dieser Stelle könnte man die Einleitung von Kapitel 3 ad absurdum führen: Solange keine konkrete Vorstellung davon existiert, *wo* sich die einzelnen Dinge eigentlich befinden, arbeitet zwar jeder Erläuterungsversuch im leeren Raum - solange aber nicht einmal klar ist, *welche* Dinge im Spiel sind, nützt eine Beschreibung ihrer Anordnung recht wenig. Deshalb gehen wir für einen Moment einfach

*) Ein weiterer Geniestreich, diesmal aus der Marketingabteilung von Borland: Die erste Version von *Graph* wird in praktisch unveränderter Form sowohl von Turbo Pascal 4.0 als auch von Turbo C 1.5 verwendet, die zweite gemeinsam von Turbo Pascal 5.0 und Turbo C 2.0. Die Grafiktreiber sind zwischen Turbo Pascal 4.0 und Turbo C 1.5 bzw. Turbo Pascal 5.0 und Turbo C 2.0 austauschbar, die Zeichensätze zwischen allen vier Compilern.

davon aus, daß sich alles an seinem richtigen Platz befindet,*) und beschreiben den Ablauf eines Aufrufs.

Ein Aufruf von *PutPixel*

Gegeben sei das folgende Programm, in das der Grafiktreiber EGAVGA nach einer Konvertierung über BINOBJ als »Prozedur *EGADriver*« direkt eingebunden ist:

```
uses Graph;
var GraphDriver, GraphMode, x: Integer;

procedure EGADriver; external;
{$L EGAVGA.OBJ}          { mit BINOBJ aus EGAVGA.BGI erzeugt }

begin
  x:= RegisterBGIDriver(@EGADriver);  { "Registrierung" }
  GraphDriver := Detect; InitGraph(GraphDriver, GraphMode,'');
  PutPixel(200,100,MaxColors);

  Readln;
  CloseGraph;
end.
```

Wie in der Überschrift angedeutet, lassen wir auch den Aufruf von *InitGraph* völlig außer acht, gehen davon aus, daß danach »alles initialisiert ist«, und konzentrieren uns auf den Ablauf von *PutPixel.*

Ein **Hinweis zur Version 5.0:** Obwohl das Unit *Graph* um ein rundes Dutzend Routinen erweitert wurde, lassen sich die Unterschiede innerhalb der Grafiktreiber in einem Satz zusammenfassen: sie bestehen aus einigen zusätzlichen Formalien im Dateikopf (siehe Seite 327) und einem wesentlich aufwendigeren Algorithmus zum Zeichnen von Linien, der um rund 20 Prozent schneller als sein Vorgänger arbeitet. Die folgenden Auszüge stammen aus der Version 4.0 - die Adressen der Version 5.0 sind entweder im Listing angemerkt oder im dazugehörigen Text erwähnt, soweit sie sich unterscheiden.

```
** PutPixel(200,100,MaxColors);          ** V 5.0: 0037 (wg. PUSH BP)
372E:0035 B8C800        MOV   AX,00C8
372E:0038 50            PUSH  AX              ; X-Koordinate auf den Stack
372E:0039 B86400        MOV   AX,0064
372E:003C 50            PUSH  AX              ; Y-Koordinate auf den Stack
372E:003D B80F00        MOV   AX,000F
372E:0040 50            PUSH  AX              ; Farbe auf den Stack
372E:0041 9A67167638    CALL  GRAPH:PUTPIXEL
```

*) Wie im nächsten Abschnitt gezeigt, ist dieses Postulat recht gewagt.

Innerhalb des Pascal-Programms zeigen sich keinerlei Besonderheiten: Die drei Parameter werden in der Reihenfolge ihrer Angabe auf den Stack gebracht, darauf folgt der Aufruf der »offiziellen« Routine von *Graph*.

```
GRAPH:PUTPIXEL:                                  ** V 5.0: GRAPH:16DB
3876:1667 55            PUSH   BP
3876:1668 8BEC          MOV    BP,SP
3876:166A 8B460A        MOV    AX,[BP+0A]        ; X-Koordinate
3876:166D 8B5E08        MOV    BX,[BP+08]        ; Y-Koordinate
3876:1670 8A5606        MOV    DL,[BP+06]        ; Farbe
3876:1673 BE3000        MOV    SI,0030           ; Funktionsnummer
3876:1676 FF1E5A03      CALL   FAR [035A]
3876:167A 5D            POP    BP
3876:167B CA0600        RETF   0006              ; Parameter vom Stack
```

Die Prozedur *PutPixel* innerhalb von *Graph* liest die drei übergebenen Parameter, lädt das SI-Register mit einer Funktionsnummer und führt einen FAR-Aufruf aus. Die Speicherzellen $035A..5D (in der **Version 5.0**: $003EA..ED) sind durch *Init-Graph* auf die Adresse eines *Dispatchers* (»Funktionsverteilers«) gesetzt worden, dessen Listing hier folgt:

```
;-----------------------------------
; Dispatcher im 2. Segment von GRAPH
; ----------------------------------

39FF:0000 1E            PUSH   DS
39FF:0001 0E            PUSH   CS                ; DS auf dieses Code-
39FF:0002 1F            POP    DS                  Segment
39FF:0003 56            PUSH   SI
39FF:0004 FF940F00      CALL   [SI+000F]         ; in diesem Segment
39FF:0008 5E            POP    SI
39FF:0009 1F            POP    DS                ; DS zurück
39FF:000A 2EFF2E4900    JMP    FAR CS:[0049]     ; zum Grafiktreiber

        < Tabelle mit den Startadressen der Routinen >
39FF:000F 0279 - Funktion $00 (V 5.0: 035F)
          0231 - Funktion $02 (V 5.0: 0317)
          ...
```

Der Dispatcher im zweiten(!) Codesegment von *Graph* ruft über die zuvor gesetzte Funktionsnummer (SI-Register) eine weitere Routine innerhalb dieses Segments auf; danach folgt der Sprung zum Grafiktreiber, dessen Adresse hier auf den Speicherzellen $39FF:0049..4C verzeichnet ist.

Die Unterschiede in der **Version 5.0** beschränken sich auf verschobene Adressen: Dort beginnt die Tabelle mit der Adresse $0013 innerhalb des Segments, die Startadresse des Grafiktreibers ist in den Speicherzellen CS:0070..73 verzeichnet.

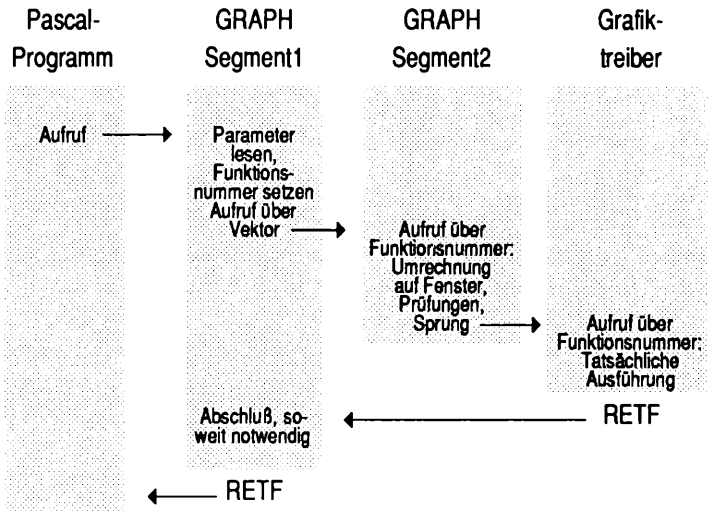
```
; ----------------------------------------
; Code für PutPixel im 2. Segment von GRAPH
; ----------------------------------------

39FF:0103 E81200        CALL   0118          ; Addition WindMin
39FF:0106 E83A10        CALL   1143          ; X/Y außerhalb des Schirms?
39FF:0109 7301          JNB    010C
39FF:010B C3            RET
39FF:010C E94BFF        JMP    0075          ; -> POP/POP/RETF
```

Der Code im zweiten Segment von *Graph* für *PutPixel*: Die Koordinaten des zu setzenden Punktes befinden sich in den Registern AX und BX. Die Routine 0118 addiert die X- und Y-Koordinaten der linken oberen Ecke des Zeichenfensters dazu. Die Routine 1143 prüft, ob der Punkt außerhalb des Bildschirms (bzw. des Zeichenfensters) liegen würde: Falls ja, wird *PutPixel* über den Sprung zu 0075 im Hauruckverfahren (POP/POP/POP/RETF) beendet. Ansonsten geht es nach dem RET-Befehl innerhalb des Dispatchers mit dem Sprung über CS:0049 weiter, der zum Grafiktreiber (»Prozedur« *EGADriver* bzw. *.BGI-Datei) führt.

Auch hier erschöpfen sich die Unterschiede in der **Version 5.0** in leicht erhöhten Adressen der beiden aufgerufenen Routinen (Addition von *WindMin*: $015C anstelle von $011D, X/Y-Prüfung: $113D) - Ablauf und Code sind derselbe.

```
; ----------------------------------------
; Dispatcher im Grafiktreiber ("EGADriver")     ; Versionen 4.0 und 5.0
; ----------------------------------------
373E:0000 1E            PUSH   DS
373E:0001 0E            PUSH   CS             ; DS auf dieses Codesegment
373E:0002 1F            POP    DS
373E:0003 FC            CLD
373E:0004 55            PUSH   BP
373E:0005 FF941600      CALL   [SI+0016]      ; das ist immer noch
373E:0009 5D            POP    BP               dieselbe Funktionsnummer!
373E:000A 1F            POP    DS
373E:000B CB            RETF
   ....
373E:0016  < Tabelle mit den Startadressen der Routinen >
```

Der Grafiktreiber beginnt mit einem weiteren Dispatcher: In unserem Beispiel enthalten AX und BX die (mittlerweile an das Zeichenfenster angepaßten) Koordinaten des Punktes, im SI-Register befindet sich nach wie vor die zu Anfang von GRAPH:PUTPIXEL gesetzte Funktionsnummer ($0030). Die hier aufgerufene Routine führt die Operation »Pixel setzen« endgültig aus. Grafisch läßt sich der Ablauf ungefähr so darstellen:



Das erste Segment von *Graph* enthält die »Logik«: Hier werden übergebene Parameter gelesen, Funktionsnummern gesetzt und eventuell mehrere Aufrufe des zweiten Segments ausgeführt (wie z.B. »Grafik-Cursor ermitteln«, »Linie zeichnen«, »Grafik-Cursor neu setzen«).

Das zweite Segment von *Graph* beginnt mit einer Sprungtabelle über die Funktionsnummer. Es enthält eine Reihe von Operationen und Prüfungen, die für eine Vielzahl von Grafikbefehlen ausgeführt werden müssen (wie Addition der Fenstergrenzen und die Prüfung, ob ein Punkt außerhalb des Bildschirms liegt).

Der Grafiktreiber beginnt ebenfalls mit einer Sprungtabelle über die Funktionsnummer und führt die eigentliche Operation aus.

Einige weitere Einzelheiten, Schlußfolgerungen und Behauptungen*):

- Der Vektor »vom ersten zum zweiten Segment« ($035A bzw. $03EA) wird erst von *InitGraph* auf den Dispatcher gesetzt - der Initialisierungsteil von *Graph* und *RestoreCrtMode* setzen ihn auf eine Pascal-Routine, die sich am Anfang des ersten Segments von *Graph* befindet:

```
procedure BGIError;
begin
  if not GraphInitialized then
     Writeln('BGI Error: Graphics not initialized (use InitGraph)')
   else Writeln('BGI Error: Not in graphics mode');
```

*) ... quod erit demonstrandum. Aber: *Graph* ist im Vergleich mit den restlichen Strukturen von Turbo Pascal so komplex ausgefallen, daß ich meinem Hobby, den Leser erst einmal aufs Glatteis zu führen, hier scharfe Zügel angelegt habe.

```
    Halt(1);
  end;
```

Auf diese Weise haben die Entwickler die Prüfungen »Grafiktreiber geladen« und »Grafikmodus gesetzt« realisiert, die bei jeder Grafikroutine ausgeführt werden müssen. (Oder auch nicht: Routinen wie *GetFillPattern*, die von der Logik her auch nach einer Umschaltung in den Textmodus sinnvolle Ergebnisse liefern müßten, brechen ebenfalls mit einer Fehlermeldung ab).

- Der Vektor »von *Graph* zum Grafiktreiber« (CS:$0049 bzw. CS:0070) macht das Programm unabhängig davon, ob sich der Grafiktreiber innerhalb des Segments CODE oder auf dem Heap befindet.
- Die Verwendung von »Funktionsnummern*)« und Sprungtabellen beseitigt eine ganze Reihe kleinerer Probleme:

  • gleiche oder ähnliche Prüfungen innerhalb von *Graph*, die für mehrere Routinen durchgeführt werden müssen, lassen sich platzmäßig optimieren;

  • Aufrufe von Grafiktreiber-Routinen sind unabhängig vom momentan geladenen Treiber. (Je nach Video-Adapter ist eine Operation wie das Setzen eines Pixels eine mehr oder weniger komplexe Angelegenheit - deshalb ergeben sich für jeden Treiber andere Startadressen der einzelnen Routinen).
- Die in den Sprungtabellen gespeicherten Startadressen sind *nicht relozierbar*. Das wäre auch nicht sonderlich sinnvoll, weil *.BGI-Dateien während des Programmlaufs geladen werden können und ein Turbo Pascal-Programm (im Gegensatz zu DOS) keinen »Relocator« enthält, der solche Symbole an ihre tatsächlichen Adressen anpaßt. Daraus ergibt sich die Forderung, daß Grafiktreiber ein eigenes Codesegment verwenden und innerhalb dieses Segments mit einer bestimmten Adresse (0000) beginnen müssen - und zwar unabhängig davon, ob sie Teil des Programms sind oder auf den Heap geladen werden. (Auf diesen Punkt kommen wir später noch zurück).

So klar, übersichtlich und flexibel diese Struktur letztendlich auch aussieht - sie enthält denselben Überbau für komplexere Operationen (Linien, Kreise, Polygone und Flächenfüllungen) wie für das simple Zeichnen eines einzelnen Punktes. Es ist nachgerade erstaunlich, daß letzteres *trotzdem* noch mit akzeptabler Geschwindigkeit vonstatten geht.

*) Eigentlich handelt es sich dabei um Indexe in den jeweiligen Sprungtabellen, die der Einfachheit halber gleich ausgerechnet sind: Die erste Funktion hat die »Nummer« $00, die zweite die »Nummer« $02, die dritte $04 usw. - ungeradzahlige »Funktionsnummern« gibt es also nicht. Die Entwickler haben sich damit die sonst übliche Operation »Funktionsnummer * Länge eines Tabelleneintrags« gespart - hier sozusagen auf unsere Kosten: die korrekte Bezeichnung wäre »Funktion $18, die über SI = $30 aufgerufen wird«.

Die Lage von *Graph* und Grafiktreibern im Speicher

Wie in Kapitel 1 angedeutet und zuvor bereits teilweise demonstriert, unterscheidet sich *Graph* in mehreren Punkten von den restlichen Standard-Units:

- *Graph* hat bereits in der Version 4.0 die Unart, eine Reihe typisierter Konstanten zu deklarieren, die folglich im Datensegment *vor* den Variablen des Pascal-Programms gespeichert werden (vgl. Seite 19). In der Version 5.0 hat sich daran nichts geändert - nur daß dort die typisierten Konstanten von *System* für eine weitere Verschiebung der Programmvariablen sorgen.
- *Graph* arbeitet mit (mindestens) zwei verschiedenen Codesegmenten, obwohl der Gesamtumfang des Codes die magische Grenze von 64 KByte nicht überschreitet*). Wer hier auf mehrere Units tippt, liegt nur teilweise richtig: *Graph* ist ein kunterbuntes Gemisch aus Pascal und »Maschine«.

Ein **Hinweis:** Die Version 4.0 von SYMDEB scheint irgendetwas gegen das Unit *Graph* (in beiden Turbo Pascal-Versionen) zu haben: beim Versuch, ein Programm GRTEST.EXE (das *Graph* enthält) mit einem der Befehle

```
SYMDEB grtest.exe
SYMDEB grtest.sym grtest.exe
```

in den Speicher zu laden, meldet sich SYMDEB mit `Processor is [80x86]` - und dann in einigen Fällen überhaupt nicht mehr. Wenn man SYMDEB dagegen ohne den Namen der .EXE-Datei aufruft (d.h. entweder mit `SYMDEB` oder mit `SYMDEB grtest.sym`) und GRTEST.EXE danach mit `N grtest.exe` und `L` separat einlädt, funktioniert die Geschichte. Welche Animositäten dabei im Spiel sind, war nicht genau zu klären - DEBUG und Periscope kennen diese Probleme jedenfalls nicht.

Dynamische Belegung über *InitGraph*

Wie in Kapitel 23 des Referenzhandbuchs angegeben, sucht *InitGraph* einen zur Hardware passenden Treiber, belegt über einen Aufruf via *GraphGetMemPtr* Platz auf dem Heap und lädt den Treiber dort hinein. Für ein Minimalprogramm, das nur die allernotwendigsten Dinge deklariert, also z.B.

```
program PlotDotDynamic;
uses Graph;
var
  GraphDriver, GraphMode, x: Integer;

begin
  GraphDriver := Detect; InitGraph(GraphDriver, GraphMode,'');
  PutPixel(200,100,MaxColors);
```

*) Da der Grafiktreiber grundsätzlich in einem eigenen Segment gespeichert ist, springt der Prozessor bei der Ausführung der meisten Grafikbefehle also in drei verschiedenen Segmenten herum.

```
  Readln;
  CloseGraph;
end.
```

ergibt sich *nach* dem Aufruf von *InitGraph* die folgende Struktur*):

$A000:0000		RAM-Ende (für 640 K)
$9000:FFFF	....	**Segment HEAP**
$9000:0000	Untergrenze der Fragmentliste	
	....	
$3A99:0003	**Grafiktreiber**	← hierher zeigt *HeapPtr*
$3858:0000		← hierher zeigt *HeapOrg*
$3458:4000	Stackbereich ($4000 Bytes)	← hierher zeigt SP
$3458:0000	... weitere SYSTEM-Variablen	**Segment STACK**
	Variable *Output* (immer def.)	
	Variable *Input* (dito)	
	... weitere GRAPH-Variablen	
	GraphGetMemPtr	
	GraphFreeMemPtr	
	... weitere Programmvariablen	**Segment DATA**
	Variable x	
	Variable *GraphDriver*	
	Variable *GraphMode*	
	....	
	typisierte Konstanten	
	von GRAPH	
$33F6:0000	Unit SYSTEM	**Segment SYSTEM**
$3300:0000	Unit GRAPH	**Segment <?>**
$yyyy:0000	Unit GRAPH	**Segment GRAPH**
$3006:0000	....	← $0054..5F undefiniert
$3000:0050	CALL SYSTEM:HALT	
	....	**Segment CODE** (Länge $60 Bytes)
	CALL GRAPH:@	
$3000:0000	CALL SYSTEM:@	

Die Belegung von Speicherplatz für den Grafiktreiber geschieht ohne jede Art von Tricks (und letztendlich über *System.GetMem*). Wie aus dem Diagramm hervor-

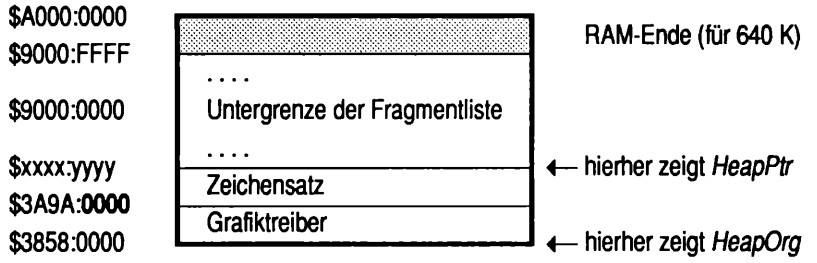
*) Die angegebenen Adressen beziehen sich auf die V 4.0 und sind auch dort nur als Richtwerte zu betrachten.

geht, wird lediglich *HeapPtr* erhöht; *HeapOrg* zeigt nach wie vor auf den Beginn des Heaps.

Ein mit

```
SetTextStyle(GothicFont, HorizDir,4);
```

aktivierter Zeichensatz wird oberhalb des Grafiktreibers auf den Heap geladen, der danach in etwa so aussieht:



Die Tatsache, daß der Grafiktreiber wie eine ganz normale dynamische Variable Platz belegt (d.h. *HeapOrg* nicht über irgendwelche Hintertüren verändert wird), bringt uns zum ersten (und erfreulicherweise auch letzten) »Vorsicht« dieses Kapitels. *Das* kann allerdings gar nicht groß genug ausfallen:

VORSICHT!!! Wer einen Zeichensatz versehentlich aus dem Heap wirft, erhält bei folgenden Ausgaben über *OutText* lediglich Undefiniertes auf dem Bildschirm. Der Grafiktreiber ist dagegen *ausführbarer Code* - ein Überschreiben seines Speicherbereichs bringt das System mit Sicherheit zum Absturz.

Wieso diese Selbstverständlichkeit einer Erwähnung wert ist? Weil in Kapitel 25 des Referenzhandbuchs steht, daß sich mit der Anweisung

```
Release(HeapOrg);
```

alle dynamischen Variablen eines Programms wieder freigeben lassen. Wenn *Release* nach einem Aufruf von *InitGraph* auf diese Weise angewendet wird, überschreiben darauffolgende Belegungen dynamischer Variablen den Grafiktreiber - und das Programm endet mit dem nächsten Aufruf einer Grafikfunktion auf recht unschöne Weise*). Dasselbe gilt für eine Anweisungsfolge wie

```
Mark(p);
...
InitGraph(...);
...
Release(p);   { wirft auch den Grafiktreiber vom Heap }
```

*) In der Version 5.0 ist die Besprechung von *Release* sowohl in Kapitel 25 als auch im Referenzteil um einen entsprechenden Hinweis erweitert.

Statische Belegung - Grafiktreiber als .OBJ-Dateien

Mit dem Zusatzprogramm BINOBJ.EXE (Turbo-Originaldiskette II) lassen sich Grafiktreiber und Zeichensätze in .OBJ-Dateien umwandeln, die danach über den Compilerbefehl {$L} statisch in ein Programm aufgenommen werden können. Wie das Kapitel 23 des Referenzhandbuchs weiterhin ausführt*), müssen die dabei entstehenden Pseudo-Prozeduren über die Funktionen *RegisterBGIFont* bzw. *RegisterBGIDriver* »registriert« werden, bevor man sie zusammen mit *InitGraph* bzw. *SetTextStyle* verwenden kann.

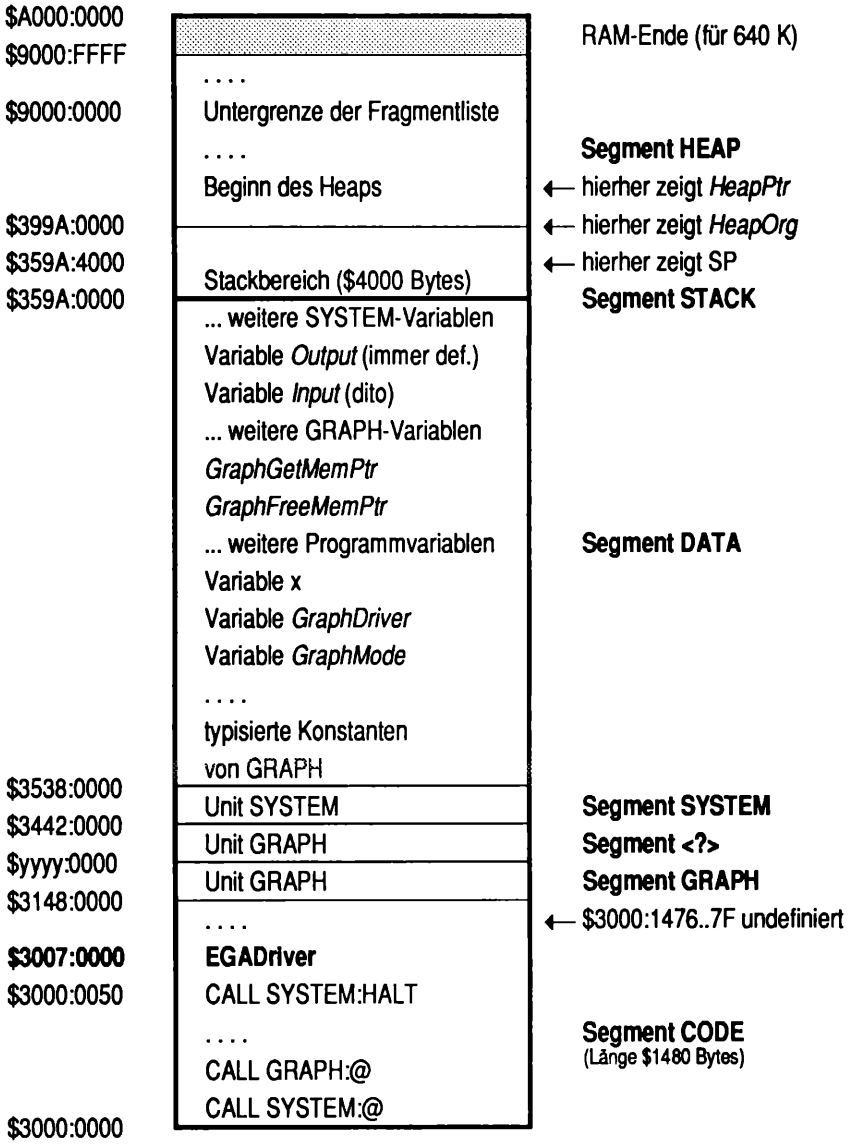
Diese Registrierung ist unter anderem wegen des »intelligenten Linkers« notwendig - in einem Programm wie

```
program PlotDotStatic;   { PLOTDOTS.PAS }
uses Graph;

var
  GraphDriver, GraphMode, x: Integer;

procedure EGADriver; external;
{$L EGAVGA}      { <== EGAVGA.BGI, über BINOBJ konvertiert }

begin
  x := RegisterBGIDriver(@EGAVGA);
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode,'');

  PutPixel(200,100,MaxColors);

  Readln;
  CloseGraph;
end.
```

wird die »Prozedur« *EGADriver* nirgendwo direkt angesprochen - der einzige Grund für den Linker, *EGADriver* nicht einfach vollständig hinauszuwerfen, ist die Adressangabe beim Aufruf von *RegisterBGIDriver* (vgl. »Interrupt-Prozeduren« in Kapitel 4).

*) ... bzw. in der Version 4.0 die Datei README auf der Diskette - sowohl BINOBJ.EXE als auch die Funktionen *RegisterBGIFont* und *RegisterBGIDriver* sind ein »last minute act«.

Das Speicherdiagramm zu *PlotDotStatic*:

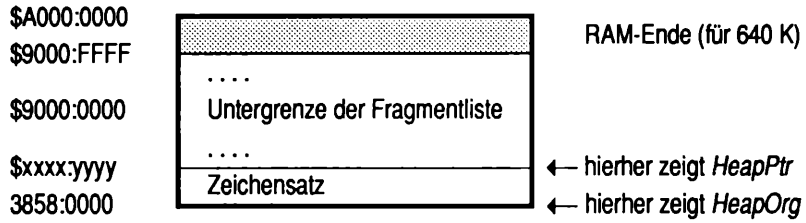


Hier befindet sich der Grafiktreiber also wie eine normale Maschinenroutine inmitten des Segments CODE. (Oder auch nicht: Wie auf Seite 318 demonstriert, arbeitet *EGADriver* mit einer eigenen Segmentadresse, die so berechnet ist, daß der erste Prozessorbefehl innerhalb der »Prozedur« die Offset-Adresse 0000 hat).

Zeichensätze, die nicht ebenfalls als .OBJ-Dateien eingebunden und über *RegisterBGIDriver* »registriert« worden sind, werden durch einen Aufruf wie

```
SetTextStyle(xxxxFont, HorizDir,4);
```

auch in diesem Fall auf den Heap geladen, der danach in etwa so aussieht:



Die Initialisierung

Graph verfügt wie alle anderen Units über einen Initialisierungsteil, der bereits durch die **uses**-Anweisung aufgenommen und zu Beginn des Pascal-Hauptprogramms aufgerufen wird.

Nun haben die Programmierer von Borland aber mehrere Variationsmöglichkeiten durch Pascal-Anweisungen vorgesehen (Festlegung/Einbinden von Treibern, Setzen von Puffergrößen etc.), die naturgemäß erst *nach* dem Startcode des Pascal-Hauptprogramms ausgeführt werden können. Deshalb (und weil die Aufnahme von *Graph* in ein Programm nicht automatisch bedeutet, daß dieses Programm beim Start einen Grafikmodus setzt), ist diese Initialisierung alles andere als vollständig - auch wenn sie eine Unmenge »namenloser« Variablen setzt, ist ihrer Besprechung in Kapitel 1 (Seite 39) nichts hinzuzufügen.

Die Hauptarbeit wird von der Routine *InitGraph* ausgeführt, ohne die *Graph* in keinem Fall auskommt. Umso verblüffender sind die beiden folgenden Feststellungen:

- *InitGraph* ist kein fester Bestandteil des Units, sondern wird »bei Bedarf« vom Linker eingebunden.
- Der Initialisierungsteil von *Graph* ist zwar offensichtlich in Pascal geschrieben, beginnt aber *nicht* mit der Adresse 0000 innerhalb des (ersten) Codesegments - und verschiebt sich bei der Aufnahme von *InitGraph* kräftig (um rund $E00 Bytes).

Vor diesem Hintergrund scheint es dann auch nicht mehr besonders wunderlich, daß in ein »Programm« wie

```
uses Graph;
begin
end.
```

insgesamt 16 Routinen von *Graph* eingebunden werden - von *SetVisualPage* (noch einsichtig) über *OutTextXY* (nicht aber: *OutText*) bis hin zu *FillPoly* (völlig unverständlich: *SetFillPattern* kommt nämlich erst zusammen mit *InitGraph*). *GetImage* ist bereits nach der uses-Anweisung mit von der Partie, *PutImage* wird dagegen auch von *InitGraph* noch nicht auf den Plan gerufen und scheint eine persönliche Einladung zu brauchen*)... ist der »intelligente Linker« bereits so intelligent, daß er nach heuristischen Prinzipen arbeitet?

Grafiktreiber im Detail

Wenn man die Datei EGAVGA.BGI mit DEBUG in den Speicher lädt und sich mit dem Dump-Befehl die ersten $90 Bytes ausgeben läßt, erhält man das folgende Ergebnis:

```
0100  70 6B 08 08
0104              42 47 49 20-44 65 76 69 63 65 20 44     BGI Device D
0110  72 69 76 65 72 20 28 45-47 41 2F 56 47 41 29 20  river (EGA/VGA)
0120  20 56 31 2E 30 30 20 2D-20 33 31 20 53 65 70 74   V1.00 - 31 Sept
0130  65 6D 62 65 72 20 31 39-38 37 0D 0A 43 6F 70 79  ember 1987..Copy
0140  72 69 67 68 74 20 28 63-29 20 31 39 38 37 20 42  right (c) 1987 B
0150  6F 72 6C 61 6E 64 20 49-6E 74 65 72 6E 61 74 69  orland Internati
0160  6F 6E 61 6C 0D 0A 00 1A                           onal....
0168                          A0 00 01 00 73 13 01 00          ....s...
0170  01 00                                            ..
0172        00 00 00 00 00 00-00 00 00 00 00 00 00 00    ..............
0180  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0190  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Die ersten beiden Bytes ($70 $6B) dienen als Kennung, werden von *InitGraph* überprüft und haben in allen .BGI-Dateien denselben Wert. Die nächsten beiden Bytes ($08 $08) haben nur einen Zweck: Sie setzen den Cursor wieder um zwei Spalten nach links - bei der Ausgabe der .BGI-Datei mit `type` werden die »Zeichen« $70 $6B also durch die folgende Copyright-Meldung überschrieben.

Die Copyright-Meldung endet mit einem Zeilenvorschub (CR/LF), einem NUL-Zeichen und danach einem Control-Z, ihre Länge ist von Treiber zu Treiber unterschiedlich.

Die nächsten 10 Bytes werden als fünf Integerzahlen interpretiert. Sie folgen in allen .BGI-Dateien direkt auf das Control-Z, ihre Position in der Datei ist also von der Länge der Copyright-Meldung abhängig. Für EGAVGA.BGI ergeben sich die Werte:

*) Falls Sie nun auf eine Erklärung warten: a) das hat sich auch in der Version 5.0 nicht geändert, b) ich habe definitiv keine.

$00A0	der Start des »eigentlichen« Dateiinhalts (relativ zum Dateianfang)
$0001	Nummer des Grafiktreibers (CGA = 0, EGAVGA = 1, HERC = 3 etc.) - nicht identisch mit den Konstanten für *GraphDriver*!
$1373	Länge des »eigentlichen« Dateiinhalts (in der Version 5.0: $1453 Bytes)
$0001	Minimalversion - wird von *InitGraph* bzw. *RegisterBGIDriver* gelesen bzw. verglichen und hat in der Version 5.0 den Wert $0002
$0001	Maximalversion - wird ebenfalls gelesen und verglichen, hat in der Version 5.0 erstaunlicherweise denselben Wert wie in der Version 4.0.

Auf diese fünf Zahlen folgen in der Version 4.0 $2E Bytes mit dem Wert 0 - in der **Version 5.0** werden dagegen vorschiedene Informationen (hauptsächlich für die neue Funktion *GetDriverName*) noch einmal wiederholt:

```
0180  A0 00 01 00 53 14 02 00-01 01 06 45 47 41 56 47   ...S......EGAVG
0190  41 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  A...............
```

Auf den Start des »eigentlichen« Dateiinhalts ($00A0), die Nummer des Grafiktreibers ($0001), die Länge des Dateiinhalts ($1453), die Minimal- und Maximalversion ($0002 bzw. $0001) folgt hier der Name des Treibers als Pascal-String, d.h. mit einem vorangestellten Längenbyte.

Danach kommt in beiden Versionen der »eigentliche« Dateiinhalt, der mit dem Funktions-Dispatcher beginnt. Dieser Dispatcher wurde auf Seite 318 (mit anderen Adressen) schon einmal gelistet:

```
01A0 1E          PUSH  DS
01A1 0E          PUSH  CS            ; DS auf dieses Codesegment
01A2 1F          POP   DS
01A3 FC          CLD
01A4 55          PUSH  BP
01A5 FF941600    CALL  [SI+0016]     ; SI = Funktionsnummer
01A9 5D          POP   BP
01AA 1F          POP   DS
01AB CB          RETF
   ....
01B6  < Tabelle mit den Startadressen der Routinen >
```

Dort wurde auch behauptet, daß der Dispatcher grundsätzlich auf der Adresse $0000 innerhalb eines Codesegments beginnen muß, weil die Laufzeitbibliothek keinen eigenen Relocator enthält und *InitGraph* die folgende Tabelle mit den Adressen der einzelnen Routinen beim Laden nicht verändert.

Folglich wird der Dispatcher über »Segmentadresse der geladenen .BGI-Datei + $000A:0000« adressiert. Wenn sich EGAVGA.BGI beispielsweise ab der Adresse $3000:0000 im Speicher befindet, dann ruft *Graph* den Dispatcher mit der folgenden Adresse auf:

```
  $3000:0000    < Startadresse der .BGI-Datei >
+       00A0    < Start des Dispatchers in der Datei >
  ----------
= $300A:0000    < Auf ganze Segmente:0000 umgerechnet >
```

Nun ist es aber eher unwahrscheinlich, daß ein Grafiktreiber so in den Speicher geladen wird, daß sich die Adresse des Dispatchers in der Form *Segment:0000* darstellen läßt (um genau zu sein: die Chancen stehen bei 1:16):

- Der Linker behandelt einen als .OBJ-Datei eingebundenen Treiber[1] wie eine völlig normale Prozedur, die »bei Bedarf« ins Codesegment des Pascal-Programms aufgenommen wird. Die Startadresse eines solchen Treibers ist deshalb von der Länge anderer Pascal-Prozeduren und Funktionen abhängig. Ein Ausschnitt aus dem Programm *PlotDotStatic* (von Seite 324) zeigt[2]:

```
** CloseGraph;
0055 9A380B7538     CALL   GRAPH:CLOSEGRAPH
** end.
005A 89EC           MOV    SP,BP
005C 31C0           XOR    AX,AX                 ; Exitcode 0
005E 9AF3016F3B     CALL   SYSTEM:HALT           ; Programmende
0063  70 6B 08 08 42 47 49 20 44 65 76 69 63 65 20 44   pk..BGI Device D
      < weitere $90 Bytes >
0103 1E             PUSH   DS        <== umgerechnet: $xxxx:0003
0104 0E             PUSH   CS
0105 1F             POP    DS
0106 FC             CLD
```

- Genau dieselbe Wahrscheinlichkeit ergibt sich beim Einladen eines Treibers in den Heap. *HeapPtr* zeigt zwar beim Start des Programms auf die Adresse $0000 innerhalb des Segments - aber eine Vorschrift, daß wir vor einem Aufruf von *InitGraph* keine dynamischen Variablen belegen dürfen, gibt es nicht:

```
....              <- HeapPtr = HEAP:0000
New(IntPtr);      <- HeapPtr = HEAP:0002
InitGraph(...)    <- lädt den Treiber ab HEAP:0002 in den Speicher
```

Zur Lösung dieser Probleme haben sich die Programmierer von Borland einen recht abenteuerlichen Ausweg einfallen lassen:

InitGraph kopiert einen Grafiktreiber erst einmal so um, daß sich die Startadresse des Dispatchers in der Form *Segment:0000* darstellen läßt. Die Verschiebung erfolgt immer in Richtung absteigender Adressen, wobei die ansonsten ungenutzten Bytes $90..$9F der .BGI-Datei als »Pufferzone« dienen.

---

[1] BINOBJ nimmt keine Veränderungen am Inhalt einer Datei vor (wie z.B. ein Herauskürzen des Copyrights).

[2] Der Linker hat anscheinend seine eigenen Ansichten über die Position eines Grafiktreibers in einer .EXE-Datei - daß sich *EGADriver* hier oberhalb des Hauptprogramms befindet, ist mehr oder weniger Zufall.

Dieser Vorgang läßt sich mit DEBUG recht gut verfolgen:

```
debug plotdots.exe
-d cs:63     (V 5.0: d cs:66)
0063  70 6B 08 08 42 47 49 20 44 65 76 69 63 65 20 44   pk..BGI Device D
       ....
      < weitere $90 Bytes >
-u 103       (V 5.0: u 106)
0103 1E              PUSH   DS       <== umgerechnet: $xxxx:0003
0104 0E              PUSH   CS
0105 1F              POP    DS
0106 FC              CLD
   .......
-g 5a        (V 5.0: g 5f)  <- bricht das Programm vor dem Aufruf von HALT ab
-d cs:63     (V 5.0: d cs:66)
0063  70 6B 08 08 42 47 49 20 44 65 76 69 63 65 20 44   pk..BGI Device D
       ....
      < weitere $8D Bytes, die letzten drei <00>-Bytes fehlen >
-u 100       (V 5.0: dieselbe Adresse!)
0100 1E              PUSH   DS       <== umgerechnet: $xxxx:0000
0101 0E              PUSH   CS
0102 1F              POP    DS
0103 FC              CLD
0104 55              PUSH   BP
0105 FF941600        CALL   [SI+0016]
```

Genauso sieht es aus, wenn ein Grafiktreiber auf den Heap geladen wird: *InitGraph* belegt zuerst dynamisch Speicherplatz, lädt dann den Treiber und verschiebt ihn schließlich so, daß die Startadresse des Dispatchers auf einer Segmentgrenze liegt. Wieso hier nicht gleich dafür gesorgt wird, daß *HeapPtr* vor dem Laden auf eine Segmentgrenze zeigt? Weil das Umschaufeln von rund 8000 Bytes zumindest von der Logik des Ablaufs her wesentlich einfacher ist als eine entsprechende »Dummy«-Variable, die nicht in allen Fällen benötigt wird.

Für Grafik-Zeichensätze gilt dasselbe - sie werden ebenfalls (und über dieselbe Routine, die auch *InitGraph* benutzt) auf eine Segmentgrenze verschoben. Allerdings beginnen hier die »eigentlichen« Daten nicht mit der Adresse $00A0, sondern mit der Adresse $0080 innerhalb der Datei. (Wir kommen in einem eigenen Abschnitt noch mehr als ausführlich auf Zeichensätze zurück).

EGAVGA.BGI - die Startadressen der einzelnen Funktionen

Das folgende Listing gibt den Funktions-Dispatcher von EGAVGA.BGI zusammen mit den Startadressen der einzelnen Routinen wieder und ist (wie alle anderen Listings von Units, Initialisierungsteilen usw.) nicht als Anleitung zum »Patchen« gedacht, sondern soll lediglich eine allgemeine Vorstellung vermitteln. Die absoluten Adressen sind von Grafiktreiber zu Grafiktreiber und von Version zu Version verschieden - solange Borland das Prinzip aber nicht völlig auf den Kopf

stellt, sollte es mit der hier gezeigten »Vorlage« aber nicht allzu schwer sein, eine beliebige Routine in einem beliebigen Treiber zu lokalisieren.

```
-----------------------------------------------
* EGAVGA.BGI - Ab Offset $A0 innerhalb der Datei
-----------------------------------------------

**************
* DISPATCHER *                                ; Aufruf mit SI = FuncNo
**************
0000 1E            PUSH   DS
0001 0E            PUSH   CS
0002 1F            POP    DS                   ; DS aufs Code-Segment
0003 FC            CLD
0004 55            PUSH   BP
0005 FF941600      CALL   [SI+0016]            ; Aufruf der Funktion
0009 5D            POP    BP
000A 1F            POP    DS                   ; DS wieder auf Original
000B CB            RETF

000C 43 42 00 00
0010 C3            RET        ******           ; Funktionen $0E,$18
0011 00 00 00 00
0015 C3            RET                         ; Funcs $06,$10,$14,$16,$18,
                                                 $2A
```

Der RET-Befehl auf der Adresse $0010 und die folgenden vier Bytes verbergen einen üblen Trick, mit dem die Programmierer von Borland die Abgrenzung zwischen den einzelnen Modulen sozusagen rückwärts durchbrechen: *InitGraph* und *SetGraphMode* schreiben hier den Befehl `CALL FAR GRAPH:$xxxx` hinein*).

```
-Tabelle mit den Startadressen der Funktionen.
      V4.0  V5.0
0016  0279  035F ; Func $00 - Mehrzweck & intern
      0231  0317 ; Func $02 -- SetGraphMode
      0257  033D ; Func $04 -- ClearDevice
      0015  0015 ; Func $06 - RET
      02E7  03D7 ; Func $08 -- MoveTo(AX,BX)
      02EF  03DF ; Func $0A -- LineTo(AX,BX)
      0303  03F3 ; Func $0C -- Line(AX,BX,CX,DX)
      0010  0010 ; Func $0E - RET
0026  0010  0010 ; Func $10 - RET
      06C0  07A4 ; Func $12 - intern
      0010  0010 ; Func $14 - RET
      0010  0010 ; Func $16 - RET
      0015  0010 ; Func $18 - RET
```

*) Dem Sinn und Zweck dieser Patcherei bin ich nicht weiter nachgegangen.

```
      00BD  0104 ; Func $1A -- SetBkColor
      00D2  012C ; Func $1C - intern
      02CF  03BF ; Func $1E -- SetColor
0036  00F8  0152 ; Func $20 -- SetFillPattern
      03C5  04B9 ; Func $22 -- SetLineStyle
      038B  047B ; Func $24 -- SetUserCharSize
      034F  043F ; Func $26 - intern
      0145  019F ; Func $28 - intern
      0015  0010 ; Func $2A - RET
      009F  00E6 ; Func $2C -- u.a. FloodFill
      0152  01AC ; Func $2E -- GetPixel(AX,BX)
0046  0162  01BC ; Func $30 -- PutPixel(AX,BX,DL)
      006C  0092 ; Func $32 -- u.a. Basisadresse für SetActivePage
      0084  00CB ; Func $34 - intern
      0099  00E6 ; Func $36 - intern
      0050  0072 ; Func $38 - intern
      ----  021C ; Func $3A - intern
```

Offensichtlich ist die Gleichung »interne Funktionsnummer von *Graph* = Funktionsnummer für den Treiber« nur in etwa der Hälfte aller Fälle zutreffend: ein guter Teil der Tabelleneinträge zeigt auf einen RET-Befehl, andere Routinen werden durch eine oder mehrere Subroutinen von *Graph* aufgerufen, die ihrerseits die Grundlage mehrerer »offizieller« Funktionen darstellen.

Die **Version 5.0** definiert den Wert $6A als höchste Funktionsnummer und deshalb eine wesentlich längere Tabelle - allerdings zeigen sämtliche Einträge für Funktionsnummern im Bereich $3C..6A auf die Adresse $0015, d.h. einen RET-Befehl. Tatsächlich gibt es in diesem Punkt also nur zwei Unterschiede:

- eine neue Funktion mit der Nummer $3A;
- und eine Verschiebung sämtlicher Startadressen um mindestens $22 Bytes (bedingt durch die längere Tabelle).

Der folgende Auszug gibt ein »typisches« Beispiel für eine Routine des Grafiktreibers wieder:

```
-----------------------------------------------------
*** Funktion $38 - liefert die Grenzen des Bildschirms
-----------------------------------------------------
0050 A3E003        MOV    [03E0],AX          ; obere linke Ecke
0053 891EE203      MOV    [03E2],BX
0057 890EE403      MOV    [03E4],CX          ; untere rechte Ecke: MaxX
005B 8916E603      MOV    [03E6],DX            MaxY
005F C3            RET
```

Das DS-Register ist momentan auf das Codesegment des Treibers gesetzt; zurückgeliefert werden die Werte von vier internen Variablen, die bei jedem Grafiktreiber an anderer Stelle gespeichert und deshalb nicht direkt für *Graph* verfügbar sind.

In der **Version 5.0** hat diese Routine die Startadresse $0072, die momentan gesetzten Bildschirmgrenzen sind ab der Adresse $04D0 gespeichert - der Code ist derselbe.

Bildformate und Parametertabellen

In diesem Punkt ist die Abgrenzung zwischen *Graph* und dem jeweils verwendeten Grafiktreiber erfreulicherweise hundertprozentig: *Graph* macht keinerlei Annahmen über die Eckwerte für einen bestimmten Videomodus - und zwar nicht nur, was die tatsächliche Adressierung von Bildpunkten betrifft, sondern auch in Bezug auf Fenstergrenzen, Funktionsnummern für den Interrupt $10, Portadressen usw.

Es gibt eine ganze Reihe von EGA-Karten, die über den üblichen Standard von 640 * 350 Bildpunkten hinausgehen, wie z.B. die GENOA HIRES oder die Paradise 480. Da der Treiber EGAVGA auch den Modus mit 640 * 480 Punkten unterstützt, sind die entsprechenden BIOS-Aufrufe und Parametertabellen bereits vorhanden - aber wie es nun einmal so ist, haben sich die einzelnen Hersteller nicht auf einen Standard zum Setzen dieser Spezialmodi einigen können. In einigen Fällen ist lediglich der Subfunktionscode für den Interrupt $10 ein anderer (wie bei der GENOA, bei der sich 640 * 480 Pixel mit `MOV AH,$73 / INT $10` setzen lassen), in anderen Fällen unterstützt die Karte die in der Werbung angepriesenen Modi mit 768 * 350 oder gar 800 * 600 Pixeln nur von der Hardware her - und der Benutzer muß sich seine Initialisierungsroutinen mit direkten Registerzugriffen selbst schreiben.

Eigene Initialisierungsroutinen sind gerade mit Turbo Pascal kein allzugroßes Problem - vorausgesetzt, man hat die entsprechenden Unterlagen - aber solange *Graph* die entsprechenden Formate nicht direkt unterstützt, ist man auch hier wieder bei den schlechten alten Zeiten angelangt:

Man schreibe eine eigene Initialisierung, eine Prozedur zum Setzen und Löschen von Punkten, vielleicht noch eine Routine zum Zeichnen von Linien und schlage sich die restlichen Möglichkeiten (wie Kreise, ausgefüllte Polygone usw. usf.) aus dem Kopf*).

Schade wär's. Zuerst einmal eine Überschrift in angemessener Größe:

*) Falls Sie stolzer Besitzer der Version 5.0 sind und im Gedenken an die neue Funktion *InstallUserDriver* gerade liebevoll Ihr Referenzhandbuch gestreichelt haben - das sollten Sie eigentlich nur dann guten Gewissens tun, wenn a) der Hersteller Ihrer Grafik-Karte einen Lizenzvertrag mit Borland abgeschlossen hat, b) der entsprechende Spezialtreiber nicht nur angekündigt, sondern auch bereits programmiert wurde, c) derselbe den Sprung über den großen Teich geschafft hat und d) Sie Ihre Grafik-Karte bis dahin nicht ausgetauscht haben - wegen hoffnungsloser Überalterung.

Eigene Bildformate für EGA-Karten mit *Graph*

Der Treiber EGAVGA.BGI unterstützt vier verschiedene Modi direkt:

- EGA mit 640 * 200 Punkten und 4 Farben
- EGA mit 640 * 350 Punkten und 16 Farben
- VGA mit 640 * 480 Punkten und 16 Farben
- EGA monochrom mit 640 * 350 Punkten

Die Ansteuerung von EGA-Registern, die Berechnung der Speicheradressen von Bildpunkten usw. ist in allen vier Fällen dieselbe - auch für den VGA-Modus. Wenn man die Monochrom-Version der EGA-Karte nicht berücksichtigt, lassen sich die Unterschiede in drei Punkten zusammenfassen: Initialisierung, maximale Y-Koordinate und Speichergröße einer Bildseite.

Der Umfang notwendiger Modifikationen und Erweiterungen eines Treibers hängt natürlich stark davon ab, wieviele der bereits vorhandenen Daten genutzt werden können:

- 640 * 480 Bildpunkte mit einer Karte, die eine eigene Initialisierung benötigt. Dieser Fall ist der einfachste, weil die notwendigen Parametertabellen bereits vorhanden sind. Solange die Karte nur eine abweichende Subfunktionsnummer für den Interrupt $10 erwartet, genügt ein Patch, ansonsten wird der zusätzliche Code entweder an die .BGI-Datei angehängt oder als Pascal-Prozedur formuliert.
- Beliebige Formate, bei denen nicht nur eine eigene Initialisierung, sondern auch eigene Adress-Ermittlungen für Bildpunkte und eigene Parametertabellen erforderlich sind.

Spezifische Initialisierungen

640 * 480 Bildpunkte für die GENOA HIRES

Die GENOA HIRES definiert ab der ROM-Version 3.0*) mehrere Subfunktionen des Interrupts $10, mit denen sich zusätzliche Bildformate setzen lassen. Der folgende Patch ist nur dann anwendbar, wenn Ihre Version von EGAVGA.BGI das Datum »2.11.87« hat - er ersetzt den Modus *VGAHi* durch den Modus »640 * 480 Pixel« der GENOA-Karte:

*) *GENOA Super EGA Programming Information and Registers' Value*, Version 1.1 vom 27.5.87, herausgegeben vom Hersteller.
Die meisten dieser Sonderformate sind nur in Kombination mit einem Multisync-Monitor verwendbar, die Subfunktionscodes liegen im Bereich von $70..$7A (AH-Register).

```
debug egavga.bgi
-u 5E8,5F0
05E8 BADF01        MOV     DX,01DF     <-- MaxY
05EB B012          MOV     AL,12       <-- Subfunktionscode für VGA
05ED BF6009        MOV     DI,0960     <-- Seitengröße in Paragraphs
05F0 EB03          JMP     05F5
-E 5EC
05EC  12.73
-w
Writing 1413 bytes
-q
```

Aktiviert wird dieser Modus mit

```
.....
GraphDriver:= Detect;  InitGraph(GraphMode, GraphDriver, '');
SetGraphMode(VGAHI);
```

oder gleich mit

```
......
GraphDriver := VGA; GraphMode := VGAHi;
InitGraph(GraphDriver, GraphMode, '');
```

Im Grafiktreiber der **Version 5.0** (EGAVGA.BGI mit dem internen Copyrightvermerk `2.00 - Mar 21 1988` und einer Dateilänge von 5363 Bytes) beginnt dieselbe Routine mit der Adresse $06DB, zu ändern ist das Byte auf der Adresse $06DF.

Für die »Tips & Tricks-Ecke« einer Computerzeitschrift wäre der Fall damit erledigt. Da die Wahrscheinlichkeit aber recht hoch ist, daß Sie entweder keine GENOA-Karte mit der ROM-Version 3.0 besitzen oder Ihre Version von EGAVGA.BGI mit abweichenden Adressen arbeitet, kommen wir auch für diesen einfachsten Fall nicht um eine kausale (und reichlich lange) Beschreibung herum:

SetGraphMode aus der Sicht des Grafiktreibers

SetGraphMode und *InitGraph* verwenden im ersten Teil ihrer Ausführung zwei Funktionen des Grafiktreibers:

$00 — wird mit dem momentan gesetzten Videomodus (*EGALo..EGAMono* = 0..3) aufgerufen und liefert die Startadresse der momentan gültigen Parametertabelle als Zeiger zurück.

$02 — wird mit dem zu setzenden Videomodus aufgerufen (Wert: 0..3) und führt die entsprechende Initialisierung durch.

Ein **Hinweis:** Wir haben bei jeder Bearbeitung einer .BGI-Datei mit DEBUG mit dem Problem zu kämpfen, daß der »eigentliche« Code des Treibers mit der Adresse $A0 in der Datei beginnt und von DEBUG ab $1A0 gelistet wird. Bei sämtlichen Adressen muß also manuell der Wert $1A0 dazuaddiert bzw. abgezogen werden - es sei denn, man behilft sich mit dem folgenden Trick:

```
debug egavga.bgi
-rcs
CS 31A5
:31bf                   <== Addition von $1A
-rds
DS 31A5
:31bf                   <== dito
-u0,5
31BF:0000 1E            PUSH    DS
31BF:0001 0E            PUSH    CS
31BF:0002 1F            POP     DS
31BF:0003 FC            CLD
31BF:0004 55            PUSH    BP
31BF:0005 FF941600      CALL    [SI+0016]

-d cs:16,19             <== Adressen der Funktionen $00 und $02
31BF:0010                    79 02-31 02                          y.1.

-u 279,281
31BF:0279 3C01          CMP     AL,01
31BF:027B 7505          JNZ     0282
31BF:027D 8B0E9001      MOV     CX,[0190]
31BF:0281 C3            RET
```

Danach erscheinen die Adressen auch mit DEBUG so, wie sie der Prozessor bei der Programmausführung sieht.

Vorsicht! Vor dem Zurückschreiben einer veränderten Datei mit dem DEBUG-Befehl `w` müssen Sie die Register CS und DS wieder auf ihre ursprünglichen Werte setzen - sonst fehlen hinterher die ersten $1A0 Bytes des Treibers.

Die folgenden Listings *sind* als Grundlage für »Patches« gedacht und gehen davon aus, daß CS und DS jeweils um $1A erhöht wurden, d.h. arbeiten wieder ohne Segmentadressen. Der Ablauf ist allen Fällen derselbe: Über das Listing des Dispatchers wird die Startadresse der Tabelle ermittelt:

```
-u0,5
0000  1E          PUSH    DS
0001  0E          PUSH    CS
0002  1F          POP     DS
0003  FC          CLD
0004  55          PUSH    BP
0005  FF941600    CALL    [SI+0016]   <== Tabellenstart: CS:0016
```

Danach wird die Startadresse der entsprechenden Funktion aus der Tabelle gelesen.

```
-d cs:16,19
0010                 79 02 31 02                    ....
```

Das Listing der Funktion $00 (Startadresse in der Version 5.0: $035F):

```
-------------------------------------------
*** Funktion $00                            ; AL = Subfunktion (0..3)
-------------------------------------------
0279 3C01          CMP    AL,01
027B 7505          JNZ    0282
027D 8B0E9001      MOV    CX,[0190]         ; siehe dritte Subfunktion
0281 C3            RET

0282 0E            PUSH   CS                ; ES aufs CSeg des Treibers
0283 07            POP    ES

-Subfunktion (AL = 2) - liefert ES:BX als Zeiger auf einen String und
 wird erst ab der Version 5.0 (für GetModeName) benutzt.
0284 3C02          CMP    AL,02
0286 7517          JNZ    029F
0288 BBBC01        MOV    BX,01BC           ; '640 x 350 EGA MONO'
028B 803E730103    CMP    Byte Ptr [0173],03  ; Modus 3 (EGAMonoHi?)
0290 740C          JZ     029E
0292 B00E          MOV    AL,0E             ; Stringlänge
0294 F6E1          MUL    CL                  * GraphMode
0296 8AD8          MOV    BL,AL
0298 32FF          XOR    BH,BH
029A 81C39201      ADD    BX,0192           ; ^string
029E C3            RET

-Subfunktion (AL <> 1, 2): CH = ??, CL = Videomodus (GRAPH).
 Liefert die Startadressen der Parametertabellen in ES:BX zurück
029F 80FD03        CMP    CH,03
02A2 750A          JNZ    02AE
02A4 C70690010100  MOV    Word Ptr [0190],0001  ; CH = 3: [0190] := 1
02AA B103          MOV    CL,03
02AC EB08          JMP    02B6
02AE 8AC5          MOV    AL,CH
02B0 32E4          XOR    AH,AH
02B2 40            INC    AX
02B3 A39001        MOV    [0190],AX         ; CH <> 3: [0190] := CH+1
02B6 C706EA030000  MOV    Word Ptr [03EA],0000  ; aktive Seite: 0
02BC 880E7301      MOV    [0173],CL         ; Videomodus (GRAPH)
02C0 8AD9          MOV    BL,CL
02C2 D0E3          SHL    BL,1
02C4 32FF          XOR    BH,BH
02C6 8B9FCF01      MOV    BX,[BX+01CF]  <== Startadressen der Tabellen
02CA 891E8E01      MOV    [018E],BX
02CE C3            RET
```

Offensichtlich ist die Arbeitsweise dieser Funktion vom Wert des AL-Registers beim Aufruf abhängig. Im Zusammenhang mit der Initialisierung ist dabei nur der Subfunktionscode 00 von Interesse:

Das CL-Register enthält in diesem Fall einen Wert, der den Konstanten *EGALo..VGAHi* entspricht (0..3). Das »Funktionsergebnis« ist ein Zeiger auf die zu diesem Modus gehörige Parametertabelle im Registerpaar ES:BX. Die Auswertung der Parametertabelle erfolgt durch die aufrufende Routine innerhalb von *Graph*.

```
; In der Version 5.0: Start mit $02B5
01CF  01D7 01EB 01FF 0213   ; Startadressen für Parametertabellen

           Pixel (hor.)                          Farbmaske
          ***       ***                            ****
01D7  0000 027F 00C7 027F 00C7 2328 1B58 1194 0808 9090   ; EGALo
01EB  0000 027F 015D 027F 015D 2328 1B58 1E46 0808 9090   ; EGAHi
01FF  0000 027F 01DF 027F 01DF 2328 1B58 2710 0808 9090   ; VGAHi
0213  0000 027F 015D 027F 015D 2328 1B58 1E46 0808 0701   ; EGAMono
                ****      ****           ****
                  Zeilenzahl       Speicherebenen-Größe?
```

Die Funktion $02 wird nicht direkt mit dem zu setzenden Videomodus aufgerufen, sondern mit einem Zeiger auf den entsprechenden Wert (auch hier: 0..3) im Registerpaar ES:BX. Unmittelbar vor dem Code der Funktion befindet sich eine Tabelle mit den Startadressen der einzelnen Initialisierungsroutinen:

```
0229 0430  043A  0448  0441   ; Startadresssen der Initialisierungsroutinen
     EGALo EGAHi VGAHi EGAMono ; In der Version 5.0: Tabellenstart $030F
---------------------------------------------
*** Funktion $02: SetGraphMode   ; V 5.0: $0317
---------------------------------------------
0231 268A07        MOV    AL,ES:[BX]          ; "GraphMode" als var-Parm
0234 240F          AND    AL,0F
0236 A27701        MOV    [0177],AL
0239 8A1E7301      MOV    BL,[0173]           ; Videomodus (GRAPH)
023D 80FB04        CMP    BL,04               ; Modi 0..3?
0240 7314          JNB    0256                ; -> nein, RET mit Carry
0242 32FF          XOR    BH,BH               ; GRAPH-Modus als Index
0244 D0E3          SHL    BL,1                  zu einer der vier Routinen:
0246 FF972902      CALL   [BX+0229]             0430, 043A, 0448, 0441
024A 8A3E7701      MOV    BH,[0177]           ; Farbe 00
024E 32C0          XOR    AL,AL               ; alle Farbregister
0250 32DB          XOR    BL,BL               ; unwichtig
0252 B410          MOV    AH,10               ; SET PALETTE
0254 CD10          INT    10
0256 C3            RET
```

Nach der Initialisierung des Video-Adapters setzt die Funktion $02 alle EGA-Farbregister auf ihre Standardwerte zurück. Ein Ergebnis im direkten Sinne hat sie nicht.

```
-XRef: Func $02, Modus = 0 (EGALo)    ; V 5.0: Start $0523
0430 BAC700          MOV    DX,00C7              ; MaxY = 199
0433 B00E            MOV    AL,0E                ; BIOS-Funktion $0E
0435 BF0004          MOV    DI,0400              ; Seitengröße $400 Paragraphs
0438 EB1B            JMP    0455                   (= 16 KByte)

-XRef: Func $02, Modus = 1 (EGAHi)    ; V 5.0: Start $052D
043A BA5D01          MOV    DX,015D              ; MaxY = 349
043D B010            MOV    AL,10                ; BIOS-Funktion $10
043F EB11            JMP    0452                 ; Seitengröße 32 KByte

-XRef: Func $02, Modus = 3 (EGAMono)  ; V 5.0: Start $0534
0441 BA5D01          MOV    DX,015D              ; MaxY = 349
0444 B00F            MOV    AL,0F                ; BIOS-Funktion $0F
0446 EB0A            JMP    0452                 ; Seitengröße 32 KByte

-XRef: Func $02, Modus = 2 (VGAHi)    ; V 5.0: Start $053B
0448 BADF01          MOV    DX,01DF              ; MaxY = 479
044B B012            MOV    AL,12        ***     ; "Patch-Stelle": Subfunc $12
044D BF6009          MOV    DI,0960              ; Seitengröße 38,4 KByte
0450 EB03            JMP    0455

0452 BF0008          MOV    DI,0800              ; Seitengröße $800 P.s (32 K)
0455 C7060C055000    MOV    Word Ptr [050C],0050 ; 80 Bytes pro Zeile
```

Wie das Listing der vier Initialisierungsroutinen zeigt, werden die Werte für *MaxX* und *MaxY*, die Größe einer Bildseite und die Anzahl der Bytes pro Zeile hier direkt gesetzt - und nicht über die Tabelle der Funktion $00. (Ob da mehrere Programmierer aneinander vorbei gearbeitet haben)?

```
; *** Version 5.0: Startadresse $054E
045B A2F103          MOV    [03F1],AL            ; Videomodus (BIOS)
045E B97F02          MOV    CX,027F              ; MaxX = 639
0461 3C0D            CMP    AL,0D                ; 320 * 200 ???
0463 7503            JNZ    0468
0465 B93F01          MOV    CX,013F              ; CX = 320 (dezimal)
0468 890EE403        MOV    [03E4],CX            ; MaxX
046C 8916E603        MOV    [03E6],DX            ; MaxY
0470 8916E803        MOV    [03E8],DX            ; MaxY
0474 C606EE0300      MOV    Byte Ptr [03EE],00   ; Annahme: 16 Farben
0479 3C0F            CMP    AL,0F
047B 7415            JZ     0492                 ; -> EGAMono
047D 3C10            CMP    AL,10
047F 7516            JNZ    0497
```

```
-- Videomodus EGAHi (BIOS $10)      ; *** V 5.0: $0574
0481 06            PUSH   ES
0482 B84000        MOV    AX,0040
0485 8EC0          MOV    ES,AX
0487 BB8700        MOV    BX,0087              ; Peek $0040:0087
048A 268A07        MOV    AL,ES:[BX]
048D 07            POP    ES
048E 2460          AND    AL,60                ; EGASize - 64/256 K
0490 7505          JNZ    0497

-- Videomodus EGAMono (BIOS $0F)    ; *** V 5.0: $0585
0492 C606EE03FF    MOV    Byte Ptr [03EE],FF   ; 4 Color Mode

-- Videomodi EGALo und VGAHi (BIOS <> $F, $10)
0497 893EEC03      MOV    [03EC],DI            ; Seitengröße (Paragraphs)
049B E80800        CALL   04A6                 ; Basisadresse Bildspeicher
049E A0F103        MOV    AL,[03F1]            ; Videomodus
04A1 32E4          XOR    AH,AH                ; Funktion 0: Set Video Mode
04A3 CD10          INT    10                   ; und Setzen
04A5 C3            RET


- Berechnung der Basisadresse des Bildspeichers  ; *** V 5.0: $0599
-Xref: $007E -SetActivePage
       $049B -Initialisierung
04A6 A1EC03        MOV    AX,[03EC]            ; Berechnung der Basisadresse
04A9 F726EA03      MUL    Word Ptr [03EA]      ; Seite * Seitengröße
04AD 0500A0        ADD    AX,A000                + EGARAM-Start
04B0 A3EF03        MOV    [03EF],AX            ; -> Basisadresse Bildspeicher
04B3 C3            RET
```

Der gemeinsame Code aller vier Initialisierungsroutinen berücksichtigt (wie auch die Adress-Berechnung für Bildpunkte) zusätzlich den Modus 320 * 200, der sich mit EGAVGA überhaupt nicht setzen läßt*).

Eigene Initialisierungen

Solange man sich auf den Modus 640 * 480 beschränkt, ist eine eigene Initialisierung auch dann nicht sonderlich problematisch, wenn die EGA-Karte keine entsprechende BIOS-Subfunktion unterstützt. Ein Ersatz für *VGAHi* läßt sich beispielsweise mit den folgenden Schritten in den Treiber einbauen:

1) Ermittlung des momentanen Dateiendes und der Adressen eventueller Variablen. Sämtliche Variablen des Grafiktreibers befinden sich im selben Segment wie der Code und haben statische Adressen, d.h. werden durch den Linker

*) Obwohl sich in der Version 5.0 einiges in EGAVGA.BGI getan hat, geistert dieser »Modus« auch hier noch im Code herum.

nicht verändert. Das DS-Register ist *nicht* durchgehend auf dieses Segment gesetzt.

2) Erstellung der Initialisierungsroutine als .COM-Datei mit einer ORG-Anweisung, die dem momentanen Dateiende der .BGI-Datei entspricht.

3) Zusammenbau von Grafiktreiber und Initialisierungsroutine mit DEBUG.

4) Veränderung des entsprechenden Startadressen-Eintrags in der Tabelle, die sich direkt vor dem Code der Funktion $02 befindet.

5) Entsprechende Erhöhung des Wertes für die Umkopierroutine von *InitGraph*. Dieser Wert befindet sich im Dateikopf vier Bytes hinter dem Control-Z der Copyright-Meldung und wird in der **Version 5.0** auf den Adressen $0184/85 wiederholt (vgl. »Grafiktreiber im Detail«).

Beliebige Bildformate

Hier kommen zu einer eigenen Initialisierungsroutine eigentlich nur zwei Punkte hinzu: eine entsprechende Veränderung der Parametertabelle (im Listing zusammen mit der Funktion $00 ab der Adresse $01CF) und eine eigene Adress-Berechnung für Bildpunkte, die sich auf ähnliche Weise an die .BGI-Datei anhängen läßt wie die Initialisierungsroutine.

Eine komplett neue Adress-Berechnung ist deshalb notwendig, weil der Wert von 80 Bytes pro Zeile, den die Initialisierung mit

```
0455 C7060C055000   MOV   Word Ptr [050C],0050  ; 80 Bytes pro Zeile
045B A2F103         MOV   [03F1],AL             ; Videomodus (BIOS)
045E B97F02         MOV   CX,027F               ; MaxX = 639
   ...
```

setzt, nur als Unterscheidung zu einem (nicht definierten) Modus mit 320 Pixeln (= 40 Bytes pro Zeile) verwendet wird.

Die Originalroutine zur Berechnung von Bildspeicher-Adressen läßt sich am einfachsten finden, wenn man die Funktion *PutPixel* (Funktionscode $30) verfolgt:

```
-----------------------------------------
*** Funktion $30: PutPixel  ; V 5.0: $01BC
-----------------------------------------
0162 8BF0           MOV   SI,AX                 ; X-Koordinate
0164 8BFB           MOV   DI,BX                 ; Y-Koordinate
0166 8AC2           MOV   AL,DL                 ; Farbe
0168 E86703         CALL  04D2                  ; Farbe 4->16, BitMask-Addr
016B 8AD0           MOV   DL,AL
016D E8DC0B         CALL  0D4C                  ; Umrechnung, Plot
0170 E97503         JMP   04E8                  ; EGA-Chip zurücksetzen
```

```
-Xref: $016D -PutPixel              ; *** V 5.0: $0E2C
0D4C E8ABF6          CALL   03FA                  ; (X,Y) (SI,DI) -> ES:DI
0D4F EB0A            JMP    0D5B

-Xref: ?? (für 320 Pixel pro Zeile)
0D51 E8A8F6          CALL   03FC                  ; (X,Y) (SI,DI) -> ES:DI
0D54 EB14            JMP    0D6A

-Xref: $0159 - GetPixel
0D56 E8A1F6          CALL   03FA                  ; (X,Y) (SI,DI) -> ES:DI
0D59 EB0F            JMP    0D6A

0D5B 52              PUSH   DX
0D5C BACF03          MOV    DX,03CF               ; Bit Mask-Register
0D5F 8AC5            MOV    AL,CH                 ; Bitmaske
0D61 EE              OUT    DX,AL
0D62 268A05          MOV    AL,ES:[DI]            ; Lesen
0D65 26880D          MOV    ES:[DI],CL            ; und Schreiben
0D68 5A              POP    DX
0D69 C3              RET

** 80 Bytes pro Zeile               ; *** V 5.0: $04EE
03FA D1E7            SHL    DI,1                  ; 40 -> 80 Bytes

-Berechnet die Adresse eines Pixels im Bildspeicher (für 40 Bytes/Zeile).
 Aufruf mit SI = X-Koordinate, DI = Y-Koordinate, DL = ??
 Zurück mit ES:DI = Adresse, CH = Bitmaske ($80/40/20..02/01), CL = Farbe

03FC 8BDF            MOV    BX,DI
03FE B102            MOV    CL,02
0400 D3E7            SHL    DI,CL                 ; DI * 4
0402 03FB            ADD    DI,BX                 ; = * 5
0404 FEC1            INC    CL                    ; CL := 3
0406 D3E7            SHL    DI,CL                 ; * 8 -> * 40
0408 8BDE            MOV    BX,SI
040A D3EE            SHR    SI,CL                 ; / 8
040C 03FE            ADD    DI,SI                 ; -> DI = Byte-Position
040E 81E30700        AND    BX,0007               ; Bitposition
0412 2E8AAFA807      MOV    CH,CS:[BX+07A8]       ; CH := Bitmaske
0417 8ACA            MOV    CL,DL                 ; CL := Farbe
0419 2EA1EF03        MOV    AX,CS:[03EF]          ; Basisadresse Bildspeicher
041D 8EC0            MOV    ES,AX
041F C3              RET
```

Die Adress-Ermittlung für 80 Bytes pro Zeile findet hier sozusagen durch eine Verdoppelung der Zeilennummer statt, die beim Aufruf im DI-Register enthalten ist.

Grafik-Zeichensätze

Im Gegensatz zu den Grafiktreibern, bei denen die Adressen einzelner Funktionen von der jeweiligen Version abhängen (und damit jedes Patch-Programm schnell zur Makulatur werden lassen), sind die Grafik-Zeichensätze durch ihre strukturelle Definition unveränderlich festgelegt - *hier* läßt sich guten Gewissens ein Programm schreiben, das direkte Modifikationen und Erweiterungen vornimmt.

Soviel als Entschuldigung dafür, warum der vorangegangene Abschnitt keinen fix und fertigen Treiber für beliebige Grafik-Karten und beliebige Versionen von EGAVGA.BGI enthält, der auf beliebigen Computern beliebige Bildformate erzeugt - sondern eben nur die Grundlage für eigene Analysen und Erweiterungen.

Das eigentliche Thema dieses Abschnitts beginnt mit einer Warnung (oder einem Trostwort - wie man will): .CHR-Dateien sind deshalb »unabänderlich in ihrer Struktur festgelegt«, weil diese Struktur außerordentlich flexibel ist - und das macht sie zum kompliziertesten Detail des gesamten Programmierpakets.

Eine Untersuchung der ersten $80 Bytes von TRIP.CHR mit DEBUG zeigt anfänglich recht wenige Unterschiede zwischen Grafiktreibern und Grafik-Zeichensätzen:

```
0100  50 4B 08 08 42 47 49 20-54 72 69 70 6C 65 78 20  PK..BGI Triplex
0110  66 6F 6E 74 20 20 56 31-30 30 20 2D 20 31 39 20  font  V100 - 19
0120  4F 63 74 6F 62 65 72 20-31 39 38 37 0D 0A 43 6F  October 1987..Co
0130  70 79 72 69 67 68 74 20-28 63 29 20 31 39 38 37  pyright (c) 1987
0140  20 42 6F 72 6C 61 6E 64-20 49 6E 74 65 72 6E 61   Borland Interna
0150  74 69 6F 6E 61 6C 0D 0A-00 1A
015A                                 80 00 54 52 49 50  tional......TRIP
0160  C9 1B 01 00 01 00 00 00-00 00 00 00 00 00 00 00  I...............
0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Auf die beiden Kennbytes $50 $4B, die für alle Grafik-Zeichensätze dieselben sind (und sich von Grafik*treibern* unterscheiden), folgen hier ebenfalls zwei Bytes mit dem Wert $08 - sie sorgen dafür, daß die ersten beiden »Zeichen« bei einer Ausgabe mit `type` durch die Copyright-Meldung überschrieben werden. Die Copyright-Meldung endet mit einem Zeilenvorschub, gefolgt von einem NUL-Zeichen und einem Control-Z. Die Position der darauffolgenden 12 Bytes innerhalb der Datei ist von der Länge der Copyright-Meldung abhängig und von Zeichensatz zu Zeichensatz unterschiedlich. Ihre Bedeutung:

`$0080`	Startadresse des »eigentlichen« Dateiinhalts (»DATAOffs«)- im Gegensatz zu Grafiktreibern, bei denen der Code mit der Adresse $00A0 beginnt
`TRIP`	Dieser Name wird von *SetTextStyle* überprüft und besteht bei jedem Zeichensatz aus vier Zeichen
`$1BC9`	Länge des »eigentlichen« Dateiinhalts (= Dateilänge - DATAOffs)
`$0001`	Minimalversion, wird von *SetTextStyle* überprüft und hat in beiden Versionen des Compilers denselben Wert
`$0001`	Maximalversion, ist in beiden Versionen des Compilers ebenfalls gleich und wird beim Laden/Aktivieren des Zeichensatzes geprüft

Zeichensätze werden wie Grafiktreiber im Speicher so umkopiert, daß sich die Startadresse des »eigentlichen« Dateiinhalts in der Form *Segment:0000* darstellen läßt. Die Verschiebung erfolgt grundsätzlich in Richtung auf niedrigere Adressen, wobei die Bytes im Bereich $0070..7F (DEBUG-Adresse: $0170..7F) als »Pufferzone dienen.

Aus der Sicht eines Programms beginnt die Bytefolge

```
0180  2B 5F 00 00 20 2D 01 00-15 00 F9 00 00 00 00 00  +_.. -....y.....
0190  00 00 04 00 40 00 64 00-78 00 E2 00 20 01 A8 01  ....@.d.x.b. .(.
      .....
```

also mit der Adresse $0000 innerhalb eines Segments. Die einzelnen Werte haben folgende Bedeutung:

0000 2B	ungeklärt
0001 5F	Anzahl der Zeichendefinitionen plus 1 (»*TblSize*«)
0002 00 00	unbenutzt
0004 20	ASCII-Code des ersten Zeichens (»*MinChar*«)
0005 012D	Startadresse der Vektordefinitionen (»*VecStart*« - Ende der Breitentabelle)
0007 00	unbenutzt
0008 15	Höhe der Großbuchstaben in Pixel (für *UserCharSize* = 4)
0009 00	unbenutzt
000A F9	Unterlänge als *ShortInt* - der Wert $F9 steht für -7

Die Bytes im Bereich $000B..0F werden nicht verwendet. Mit der Adresse $0010 ($90 innerhalb der Datei) beginnt die Vektor-Adresstabelle:

0010 0000	die Definition des ersten Zeichens (*MinChar* = $20 = Leerzeichen) beginnt $0000 Bytes nach dem Ende der Breitentabelle (*VecStart* + $0000 = $012D)
0012 0004	die Definition des zweiten Zeichens (*MinChar* + 1 = $21 = Ausrufezeichen) beginnt $0004 Bytes nach dem Ende der Breitentabelle (VecStart + $0004 = $0131)

Die Vektor-Adresstabelle enthält für jedes der $5F Zeichen der Datei die Startadresse seiner Definition und endet für GOTH.CHR mit einem Eintrag auf der Adresse $00CC (DEBUG-Adresse $024C):

00CC 1A7E	die Definition des letzten Zeichens (*MinChar* + *TblSize* - 1 = $7E = Tilde) beginnt $1A7E Bytes nach dem Ende der Breitentabelle (*VecStart* + $1A7E = $1BAB)

An die Vektor-Adresstabelle schließt sich eine Zeichenbreiten-Tabelle an, die für jedes Zeichen (in derselben Reihenfolge wie bei den Vektor-Startadressen) die Breite in Pixeln enthält. Die Startadresse dieser Tabelle ergibt sich aus *TblSize* * 2 + $0010, für TRIP.CHR also als $00CC (DEBUG-Adresse $024E):

00CE 10	das Zeichen *MinChar* (= Leerzeichen) hat die Breite 16
00CF 07	das Zeichen *MinChar* + 1 (= »!«) hat die Breite 7
00D0 0E	das Zeichen *MinChar* + 2 (= »"«) hat die Breite 14

.....

Auf die Breitentabelle folgen schließlich (ab der Adresse *VecStart* = $012D, DEBUG-Adresse $02AD) die einzelnen Zeichendefinitionen. Da die jeweiligen Startadressen in der ersten Tabelle verzeichnet sind, ist die Reihenfolge dieser Definitionen nicht unbedingt an den ASCII-Code gebunden.

```
012D 90 00 00 00
```

Die Vektoren für *MinChar* (Leerzeichen): »Bewegung ohne Zeichnen um 16 Pixel horizontal«, danach Ende der Definition.

```
0131  81 15 80 94 80 92 81 8A 81 15 81 87 82 87 81 15
0141  82 95 82 87 82 15 83 94 83 92 82 8A 81 03 80 82
0151  80 81 81 80 82 80 83 81 83 82 82 83 81 83 81 02
0161  81 81 82 81 82 82 81 82 87 00 00 00
```

Und diese schier endlose Folge von Bytes ist ein Ausrufezeichen (*MinChar* + 1) - ob Sie es nun glauben oder nicht.

Vektordefinitionen

Jedes Zeichen eines Grafik-Zeichensatzes ist als Folge von X/Y-Wertepaaren definiert, wobei ein Wertepaar aus zwei aufeinanderfolgenden Bytes besteht und einen Punkt relativ zum »Ursprung« des Zeichens beschreibt. Wenn sich der Grafik-Cursor beispielsweise auf dem Punkt (100,100) befindet und *OutText* ein Zeichen mit den drei Vektorpaaren

```
4,4  6,9,  12,0
```

ausgibt, dann wird der Grafik-Cursor während der Zeichenausgabe folgendermaßen bewegt:

```
4,4   -> 104, 96  (GetX + X-Vektor, GetY - Y-Vektor)
6,9   -> 106, 91
12,0  -> 112, 100 (Endposition)
```

Die X-Werte werden hier zum (konstanten) Ursprung des Zeichens dazuaddiert, die Y-Werte dagegen abgezogen - woraus sich eine Bewegung »nach oben im Bildschirm« ergibt.

Wenn nun einfach festgelegt wäre, daß jede Bewegung Spuren in Form einer Linie hinterläßt, dann müßten die Zeichen in Form durchgehend miteinander verbundener Linien erscheinen - außerdem lassen sich Unterlängen (hier: eine Bewegung auf Y-Koordinaten > 100) mit positiven Y-Vektoren so nicht darstellen. Deshalb enthält jedes Vektorpaar neben den eigentlichen Werten zwei weitere Informationen:

```
  X-Vektor          Y-Vektor
F V x x x x x x   F V x x x x x x

F = Funktionscode, V = Vorzeichen
```

OutText bildet über die MSBs (Bits 7) eines Vektorpaars mit der Formel

```
FCode := (XVec and $80) shr 6          { MSB von X * 2 }
       + (YVec and $80) shr 7;         { + MSB von Y }
```

einen »Funktionscode« und interpretiert die entstehenden Werte (0..3) auf die folgende Weise:

LDONE [0] - Ende der Vektordefinition. Der Grafik-Cursor bleibt auf dem »momentanen« Punkt stehen.

LDUMMY [1] - eine »überflüssige« Kombination, die von *OutText* zusammen mit den angegeben Werten ignoriert wird (und meines Wissens in keinem der vier Zeichensätze vorkommt)

LMOVE [2] - bewegt den Grafik-Cursor wie zuvor beschrieben, ohne dabei zu zeichnen. Diese Operation entspricht der Routine *MoveTo* von *Graph*

LDRAW [3] - bewegt den Grafik-Cursor in derselben Weise wie LMOVE und zeichnet dabei eine Linie ab der vorherigen Position (entspricht der Routine *LineTo*)

Das Bit 6 jedes Vektors wird als Vorzeichen verwendet - wenn es gesetzt ist, behandelt *OutText* den eigentlichen Vektorwert (Bits 5..0) als negative Zahl. Im Prinzip läuft diese Interpretation auf einen *ShortInt* mit sieben Bits (also einem Wertebereich von -64..+63) hinaus.

Die vollständige Auswertung eines Vektorpaars, in Einzelschritten und mit Pascal-Anweisungen ausgedrückt:

```
< Ermittlung eines Vektorpaars (XVec, YVec) >

FCode := (XVec and $80) shr 6      { MSB von XVec * 2 }
       + (YVec and $80) shr 7;     { + MSB von YVec }
XVec := XVec and $7F;
YVec := YVec and $7F;        { Funktionscodes entfernen }

if (XVec and $40) <> 0 then   { Bit 6 gesetzt? }
begin
  XVec := XVec and $BF;      { 1011 1111 - Bit 6 löschen }
  XVec := 64 - XVec;
end;
if (YVec and $40) <> 0 then  { dito für YVec }
begin
  YVec := YVec and $BF; YVec := 64 - YVec;
end;

case FCode of
  0: < Vektordefinition zuende >
  1: ;
  2: MoveTo(XPoint + XVec, YPoint - YVec);
  3: LineTo(XPoint + XVec, YPoint - YVec);
end;
```

Auf dieser Grundlage läßt sich die im letzten Abschnitt gezeigte Vektordefinition eines Leerzeichens etwas detaillierter betrachten:

`$90 00` X-Vektor $90 = $80 + $10 => Funktionscode LMOVE, Wert $10; Y-Vektor $00 - also eine Bewegung um 16 Pixel nach rechts

`$00 00` Ende der Definition.

Die Vektorpaare des Ausrufezeichens von TRIP.CHR sind da schon etwas komplizierter. Gehen wir davon aus, daß sich der Grafik-Cursor auf dem Punkt (100,100) befindet, und setzen wir eine Skalierung von 1:1 an:

`81 15` X-Vektor 81 = 80 + 01 => Funktionscode LMOVE, Wert 01; Y-Vektor 15 - Bewegung um 1 Pixel nach rechts und um 21 Pixel nach oben. Neue Position des Grafik-Cursors: 101,79.

`80 94` X-Vektor 80 = 80 + 00; Y-Vektor 94 = 80 + 14 => Funktionscode LDRAW - Zeichnen einer Linie von der vorherigen Position (101,79) zum Punkt (100,80), also um ein Pixel nach links und ein Pixel nach oben.

`80 93` X-Vektor 80 = 80 + 00; Y-Vektor 93 = 80 + 13 => Funktionscode LDRAW - Zeichnen einer Linie von der vorherigen Position (100,80) zum Punkt (100,81), also ein Pixel senkrecht nach oben.

Bevor wir zu einem Programm kommen, das diese mühselige Kleinarbeit erledigt, noch ein paar allgemeine Dinge (anstelle der restlichen 27 Vektorpaare des Ausrufezeichens von TRIP.CHR):

- Wie sich auch an den unterschiedlichen Längen der .CHR-Dateien feststellen läßt, sind die einzelnen Zeichensätze mehr oder weniger kompliziert aufgebaut. Da die Vektorpaare ausschließlich Liniendefinitionen enthalten, müssen Kreise (wie für ein »O« oder ein »@«) aus vielen kleinen Linienstücken zusammengesetzt werden - einige der Definitionen in GOTH oder TRIP bestehen aus mehr als 60 Vektorpaaren.

- Unabhängig von der momentan gesetzten Skalierung werden grundsätzlich alle Vektoren einer Definition bearbeitet. Daß die Zeichenausgabe bei hohen Vergrößerungsfaktoren trotzdem ein gutes Stück langsamer wird, liegt ausschließlich am Zeitbedarf von *LineTo* (bzw. der Linien-Routine des Grafiktreibers, die von *OutText* direkt aufgerufen wird).

- Die verwendete Liniendicke ist ebenfalls unabhängig von der momentan gesetzten Skalierung und entspricht *NormalWidth*, d.h. alle Linien werden mit einem Pixel Breite gezeichnet. (Für jeden Zeichensatz ergibt sich deshalb so etwas wie ein idealer Größenbereich bzw. das Gegenteil davon).

- Wenn der Vergrößerungsfaktor mit *SetTextStyle* auf den Wert 4 gesetzt wurde, dann setzt *OutText* die in den Vektordefinitionen angegebenen Werte direkt in Koordinaten um. Intern entspricht dieser »Vergrößerungsfaktor« also einer Skalierung von 1.

- Alle Großbuchstaben eines Zeichensatzes haben dieselbe (und im Kopf der Datei verzeichnete) Höhe, die Zeichen »p«, »q« und »g« haben die im Dateikopf angegebene Unterlänge. Diese (willkürlich gezogene) Grenze wird nur von runden und geschweiften Klammern durchbrochen, die in den Zeichensätzen GOTH, TRIP und SANS zwischen zwei und drei Pixel über die Höhe der Großbuchstaben hinausgehen.
- Die Funktion *TextHeight* ignoriert übergebene Strings komplett*) und liefert grundsätzlich das Ergebnis »(Höhe für Großbuchstaben + Unterlänge) * Skalierungsfaktor«.

Ausgabe von Zeichendefinitionen mit einem Pascal-Programm

DRAWCHAR.PAS faßt die bisher besprochenen Operationen und Prinzipien zusammen: das Programm liest eine .CHR-Datei ein und gibt einzelne Zeichendefinitionen »zu Fuß« aus.

```
program DrawChar;  { 19-01-88 as }
{ Rudimentäre Ausgabe von Grafikzeichen "zu Fuß" }
uses Crt,      { ausschließlich wg. ReadKey }
     Graph;

const
  XPoint = 100;   { horizontaler Startpunkt für die Zeichenausgabe }
  Prompt = 'Zeichen (<ESC> = Ende): ';
var
  GraphDriver, GraphMode: Integer;
  CHRFile: file;          { Zeichensatz-Datei, untypisiert }
  FBuf: array[0..10000] of Byte;  { Puffer dafür }
  WPtr: ^Word;
  x: Integer; FName: String;

  DataOffs,       { Start der "eigentlichen" Daten }
  MinChar,        { ASCII-Code der ersten Definition }
  TBStart,        { Start der Vektor-Adresstabelle }
  TblSize,        { Anzahl der Definitionen }
  WidthTbl,       { Start der Breitentabelle }
  VecStart,       { Start der Vektortabelle }
  CUp, CDown      { Höhe für Großbuchstaben, Unterlänge }
            : Integer;
  MAG, GLine: Integer;  { Skalierungsfaktor, Grundlinie }
  Ch : Integer;       { Auszugebendes Zeichen }
  Index: Integer;     { Vektordefinition des "momentanen" Zeichens }
  CharWidth: Integer;  { Breite des "momentanen" Zeichens }
```

*) Im Gegensatz zur Funktion *TextWidth*, die die entsprechenden Einträge der Breitentabelle aufaddiert.

Die Dateivariable für die .CHR-Datei ließe sich genauso gut als *file of Byte* deklarieren - nur dauert dann das Einlesen wesentlich länger. Die Größe von *FBuf* sollte auch für alle zukünftigen Zeichensätze ausreichend sein (korrekterweise hätte man hier eine Zeigervariable definieren und abhängig von der tatsächlichen Dateigröße Platz auf dem Heap belegen sollen). *WPtr* wird zum Auslesen von Integerwerten aus *FBuf* verwendet und erspart Zuweisungen der Form *Int_Var := FBuf[x] + 256 * FBuf[x+1];* (vgl. »Variablengrößen und -strukturen« in Kapitel 3).

```
procedure PaintChar(Index: Integer);
const
  LDONE = 0; LMOVE = 2; LDRAW = 3;  { die drei Funktionstypen }
var
  XVec, YVec, Func: Integer;
begin
  repeat
    XVec := ShortInt(FBuf[Index]);     { erster/nächster Vektor }
    YVec := ShortInt(FBuf[Index+1]);
    Inc(Index,2);
      { MBS = Funktionscodes }
    Func := (XVec and $80) shr 6 + (YVec and $80) shr 7;
    XVec := XVec and $7F; YVec := YVec and $7F;  { Funktionscodes raus }

    if XVec >= $40 then XVec := -128 + XVec;     { Bit 6: Vorzeichen }
    if YVec >= $40 then YVec := -128 + YVec;

    XVec := XVec * MAG; YVec := YVec * MAG;      { Skalierung }
    case Func of
      LMOVE:   { Bewegung ohne Zeichnen }
             MoveTo(XPoint+XVec,GLine-YVec);
      LDRAW:  { Bewegung mit Zeichnen }
             LineTo(XPoint+XVEc,GLine-YVec);
       end;
  until Func = LDONE;
end;
```

Die »Kernroutine« von *DrawChar* wird mit dem Index zum ersten Vektorpaar des gewünschten Zeichens als Parameter aufgerufen. Von ihrem theoretischen Gegenstück unterscheidet sie sich nur in einer Kleinigkeit: Anstatt bei negativen Vektorwerten erst das gesetzte Bit 6 zu löschen, können wir den (durch dieses »Vorzeichen« um 64 erhöhten) Wert auch einfach von -128 (anstelle von -64) abziehen.

PaintChar verhält sich in jedem Punkt so wie die Zeichenroutine von *OutText*: Der Funktionscode 01 wird ignoriert, nach dem Lesen des Funktionscodes 00 bleibt der Cursor auf seiner vorherigen Position stehen. Eine Begrenzung irgendwelcher Art existiert nicht - die Schleife wird unerbittlich solange durchlaufen, bis *Index* irgendwann ein Vektorpaar mit dem Funktionscode 00 zutage fördert.

```
  { ---------------------------------------- }
begin
  { 1. Eingabe des Dateinamens und Einlesen der Datei in FBuf }
  if ParamCount = 0 then
  begin
    Write('Zeichensatz (GOTH, TRIP, SANS, LITT)? ');
    Readln(FName);
  end
  else
    FName := ParamStr(1);
  if Pos('.',FName) = 0 then FName := FName + '.CHR';
  Assign(CHRFile, FName);  {$I-} Reset(CHRFile,1); {$I+}  { Länge 1 }
  If IOResult <> 0 then
  begin
    Writeln('Nicht gefunden: ', FName); Halt;
  end;
  BlockRead(CHRFile,FBuf,FileSize(CHRFile));
  Close(CHRFile);
```

DrawChar läßt sich entweder mit dem Namen der .CHR-Datei aufrufen oder fragt selbständig nach. In beiden Fällen wird die Endung .CHR notfalls automatisch an den Dateinamen angehängt.

```
  { 2. Ermittlung der Eckdaten, die in der .CHR-Datei gespeichert sind }
  x:= 0;
  while (x < $80) and (FBuf[x] <> $1A) do Inc(x); { Suche nach ^Z }
  if x = $80 then
  begin
    Writeln('Datei ', FName, ' nicht im erwarteten Format!');
    Halt;
  end;
  Inc(x);
  WPtr := @FBuf[x];  DataOffs := WPtr^;  { Start des "eigentlichen" Inhalts }
  TblSize := FBuf[DataOffs + 1];  { Anzahl der Zeichendefinitionen }
  MinChar := FBuf[DataOffs + 4];  { niedrigster ASCII-Code }
  CUp := FBuf[DataOffs+8];        { Höhe für Großbuchstaben }
  CDown := ShortInt(FBuf[DataOffs+$0A]);  { Unterlänge }
  TBStart := DataOffs + $10;      { Tabellenstart }
  WidthTbl := TBStart + TblSize shl 1;  { Start der Breitentabelle }
  WPtr := @FBuf[DataOffs+5];
  VecStart := WPtr^ + DataOffs;   { Start der Vektordefinitionen }
```

Bei der Suche nach dem abschließenden Control-Z wurde eine Sicherung eingebaut - falls sich dieses Zeichen innerhalb der ersten $80 Bytes der Datei nicht finden läßt, bricht das Programm mit einer entsprechenden Meldung ab. (Hier gibt es eine ganze Reihe von Möglichkeiten zur Verifizierung: *SetTextStyle* und *RegisterBGI-Font* prüfen die beiden Kennbytes am Dateianfang, enthalten aber keine weiteren Abbruchbedingungen; das im übernächsten Abschnitt vorgestellte Programm CE

vergleicht die beiden nach der Copyright-Meldung folgenden Parameter mit der Dateilänge).

Der Beginn der »eigentlichen« Daten (Variable *DataOffs*) muß hier bei jeder Indizierung dazugerechnet werden, weil wir in einer Datei natürlich keine Segmentadressen entsprechend umsetzen können. (Das hätte sich allerdings auch über einen Zeiger auf ein Array lösen lassen, der auf die Adresse von *FBuf[DataOffs]* gesetzt wird).

```
{ 3. Grafikmodus und Skalierung setzen, Ausgabe einzelner Zeichen }
GraphDriver := Detect; InitGraph(GraphMode, GraphDriver, '');
if GraphDriver = CGA then SetGraphMode(CGAC1);  { 320 * 200 Pixel für CGA }

MAG := GetMaxY div (CUp - CDown + 8);
GLine := (CUp +6) * MAG;   { Grundlinie }
```

Das Programm verwendet einen Skalierungsfaktor (Variable *MAG*), der abhängig von der Zeilenzahl des Bildschirms und dem Platzbedarf der Zeichendefinitionen gesetzt wird. Auf diese Weise sind wir einigermaßen unabhängig vom tatsächlich verwendeten Video-Adapter.

```
repeat
  { 3a. Zeicheneingabe }
  SetColor(White);
  MoveTo(0,0);
  OutText(Prompt);  { 'Zeicheneingabe ... ' }
  repeat
    Ch := Ord(ReadKey);
    if Ch = 27 then
    begin
      CloseGraph;
      Halt;          { Programmende }
    end;
  until (Ch >= MinChar) and (Ch <= MinChar + TblSize - 1);
```

Die Eingabe des Zeichens, das von *DrawChar* vergrößert dargestellt werden soll: Es werden nur Zeichen akzeptiert, die auch tatsächlich in der .CHR-Datei definiert sind*), mit ESC läßt sich das Programm abbrechen.

```
  { 3b. Grundlinie }
  ClearViewPort;   { ganzen Bildschirm löschen }
  MoveTo(0,0);
  OutText(Prompt + Chr(Ch));   { 'Zeicheneingabe: ' + Zeichen }
  SetColor(Red);
  Line(XPoint,20,XPoint,GetMaxY);    { Linker Rand }
  Line(XPoint,GLine,GetMaxX,GLine); { Grundlinie }
  SetColor(White);
```

*) Ganz im Gegensatz zu *OutText* - siehe nächster Abschnitt.

```
      { 3c. Lesen der Vektordefinition und Zeichnen }
      WPtr := @FBuf[TBStart + (Ch - MinChar) * 2];
      Index := WPtr^ + VecStart;  { Start-Index der Vektordefinition }
      PaintChar(Index);  { Zeichnen des Zeichens }

      { 3d. Rechter Rand }
      CharWidth := FBuf[WidthTbl+ Ch - MinChar] * MAG;  { Zeichenbreite }
      SetColor(Red);
      Line(XPoint+CharWidth,20,XPoint+CharWidth,GLine);  { rechter Rand }

    until false;
  end.
```

DrawChar verwendet dieselbe Methode für die Ermittlung des rechten Randes wie *OutText*, ignoriert die nach *PaintChar* »stehengebliebene« Position des Grafik-Cursors und greift auf den entsprechenden Eintrag der Zeichenbreiten-Tabelle zu. Ein schlichter Aufruf von *LineTo* mit der von *PaintChar* gesetzten Cursorposition als Parameter, also

```
LineTo(20,GetY);
```

hätte dasselbe Ergebnis wie die Berechnung über *CharWidth*, weil jede Zeichendefinition nicht nur mit dem »rechten Rand« endet, sondern den Grafik-Cursor auch wieder auf die Grundlinie setzt. (Wenn das nicht der Fall wäre, würden folgende Zeichenausgaben vertikal versetzt erscheinen).

Ein Tip: Probieren Sie *DrawChar* einmal zusammen mit SANS.CHR aus und geben Sie irgendein Steuerzeichen (z.B. RETURN) ein - da haben sich die Programmierer von Borland offensichtlich einen kleinen Scherz geleistet.

Die Erweiterung von Zeichensätzen

Nehmen wir das größte Problem gleich vorweg: Deutsche Umlaute fallen offensichtlich seit neuestem[1] für amerikanische Programmierpakete nicht mehr in die Klasse »Exotika, unwichtig«, sondern in die Kategorie »Unerwünscht«. Eine Befehlsfolge wie

```
SetTextStyle(TriplexFont,HorizDir,4);
OutText('Größe');
```

erzeugt nicht etwa einen Leerraum anstelle des »öß«, sondern Undefiniertes für das »ö« - und ein »a« anstelle des »ß«[2]. Der Grund: Solange ein anderer Zeichensatz

[1] So neu ist diese Klassifikation allerdings nicht: Der INPUT-Befehl von Turbo BASIC behandelt die Umlaute ebenfalls als unerwünschte Grafikzeichen - und würde das auch in der deutschen Version tun, wenn ich den Compiler nicht Anfang 1987 entsprechend gepatcht hätte (einen Tusch, bitte...)

[2] ... und die einmalige Übersetzung von *Size* und *User defined size* als »Faktor« bzw. »Vom Benutzer definiert!«, weil ich mich in der 4.0-Version von GRDEMO (Originaldiskette III) bei allen Ausgaben über Vektorzeichen-

als *DefaultFont* gesetzt ist, wirft *OutText* an irgendeiner Stelle das MSB des übergebenen ASCII-Codes (mit AND $7F) einfach hinaus - aus dem ASCII-Code $E1 (ß) wird damit der Code $61 (a). Dieselbe Operation mit einem »ö« (ASCII-Code $94) durchgeführt, ergibt $14 - und weil davon zu einem späteren Zeitpunkt der Wert von *MinChar* ($20) ohne weitere Prüfungen abgezogen wird (Ergebnis $F4), liest *OutText* zwei Einträge der Zeichenbreiten-Tabelle als »Startadresse der Vektordefinition«.

Ergo: Wenn man nicht gerade anfangen will, in GRAPH.TPU herumzupatchen, ist die Erweiterung von Grafik-Zeichensätzen über den ASCII-Code $7F hinaus unmöglich. Es bleiben zwei Auswege:

- Ersatz einzelner Definitionen (wie in den schlechten alten Zeiten von CP/M: aus den eckigen Klammern und »\« wird »ÄÖÜ«, aus den geschweiften Klammern und »|« wird »äöü«);
- Erweiterung *nach unten*. Alle Zeichensätze außer SANS definieren das Leerzeichen als niedrigsten ASCII-Code - und da dieser Code als *MinChar* in der Datei selbst gespeichert ist, lassen sich bis zu 32 weitere Definitionen in jeder .CHR-Datei unterbringen, ohne daß sich dabei Inkompatibilitäten ergeben. (SANS.CHR macht diese Möglichkeit sozusagen vor: In dieser Datei hat *MinChar* den Wert 1, der Eintrag für das Leerzeichen steht in der Vektor-Adresstabelle und der Zeichenbreiten-Tabelle erst an 32. Stelle. Die ersten 31 Einträge zeigen sämtlich auf dieselbe Definition - eine Spielkarte. Ähnlich sieht es mit GOTH.CHR in der Version 5.0 aus - dort sind die ASCII-Codes 0..31 als Kreuzzeichen definiert).

Unabhängig davon, welches Verfahren verwendet wird[3], brauchen wir in jedem Fall einen Ersatz für *OutText* und *OutTextXY*, der in etwa so aussieht:

```
procedure OutText(s: string);
const
  NewCodes: array[1..8] of Char = (#xx, #yy, #zz ... );
  OldChars: array[1..8] of Char = ('Ä','Ö','Ü', #21, 'ä', 'ö', 'ü', 'ß');
var x,y: Integer;                                { '§' }
begin
  for x:= 1 to Length(s) do
    for y:= 1 to 7 do
     if s[x] = OldChars[y] then s[x] := NewCodes[y];
  Graph.OutText(s);
end;
```

Ein **Hinweis:** Weil die Absuche von Strings mit Pascal-Anweisungen unter Umständen einige Zeit dauert, werden auf Seite 400 einige Alternativen dazu vorgestellt.

sätze um die Umlaute herumdrücken mußte. In der Version 5.0 ist GRDEMO von Heimsoeth in Eigenregie eingedeutscht worden - ich bin gespannt, was die daraus gemacht haben.

[3] Das im nächsten Abschnitt vorgestellte Programm CE kann Definitionen sowohl ersetzen als auch neu anfügen.

Anfügen und Löschen von Zeichendefinitionen

Das Anfügen einer weiteren Zeichensatzdefinition ist im Prinzip eine recht geradlinige Angelegenheit (wenn man von der Menge der Details absieht):

Die Vektor-Adresstabelle muß um einen Eintrag erweitert werden, der Rest der Datei verschiebt sich entsprechend um zwei Bytes nach hinten. Wohin dieser Eintrag zeigt, spielt keine allzu große Rolle: Weil *OutText* die Adressen über die (nach ASCII-Codes geordnete) Vektor-Adresstabelle ermittelt, können die Vektordefinitionen selbst in beliebiger Reihenfolge gespeichert sein. Die Zeichenbreiten-Tabelle muß ebenfalls um einen Eintrag erweitert werden (womit sich der Rest der Datei um ein weiteres Byte verschiebt). Die Gesamtzahl der Zeichendefinitionen (*TblSize*) wird um eins, der Zeiger auf die erste Vektordefinition (*TBStart*) um drei heraufgesetzt, *MinChar* dagegen um eins erniedrigt. Praktischerweise kommt die neue Vektordefinition ans Ende der Datei, deren Länge sich entsprechend erhöht. Als letztes muß der Eintrag für die Anzahl der von *SetTextStyle* zu verschiebenden Bytes (hinter der Copyright-Meldung) an die neue Dateilänge angepaßt werden.

Erweiterungen sind immer nur »nach unten« möglich, weil sich durch das *Einfügen* eines neuen Eintrags der Zusammenhang zwischen ASCII-Codes und Tabelleneinträgen verschieben würde.

Das Löschen einer Definition macht diese Schritte mehr oder weniger rückgängig, muß aber auch inmitten der Tabelle möglich sein (weil sich sonst die Operation »Ersetzen« nicht realisieren läßt):

- Wenn der zu löschende Eintrag am Anfang oder Ende der Tabelle liegt, läßt sich *MinChar* herauf- bzw. *TblSize* herabsetzen, die entsprechenden Einträge in der Vektor-Adresstabelle und der Zeichenbreiten-Tabelle sowie die Vektordefinition selbst können komplett entfernt werden.

  Ansonsten wird die Vektordefinition auf ein einzelnes LDONE-Paar (mit dem Wert 00 00) verkürzt, der Eintrag in der Vektor-Adresstabelle bleibt unverändert (und zeigt danach auf dieses 00 00-Vektorpaar).

  Wie die Datei SANS.CHR demonstriert, können mehrere Einträge in der Vektor-Adresstabelle auf ein und dieselbe Vektordefinition zeigen: Wenn das der Fall ist, wird die Datei durch eine Löschung um zwei Bytes *vergrößert*, weil wir ein Vektorpaar mit den Werten 00 00 ans Ende anfügen und den »gelöschten« Eintrag der Vektor-Adresstabelle auf dieses Paar setzen müssen.

- Solange die Vektor*definition* nicht gerade am Dateiende steht, verschieben sich alle folgenden Vektordefinitionen nach vorne und machen eine Korrektur der entsprechenden Einträge in der Vektor-Adresstabelle erforderlich*).

*) »Folgende Definitionen« bezieht sich in diesem Ausnahmefall nicht auf Zeichen mit höheren ASCII-Codes, sondern auf Definitionen mit höheren Speicheradressen.

Der *Ersatz* einer Zeichendefinition läßt sich am einfachsten realisieren, indem man das Zeichen zuerst löscht (d.h. auf 00 00 reduziert oder ganz entfernt) und dann eine neue Definition anfügt.

Damit hätten wir die Grundlage für einen Editor abgehandelt, bei dem der Benutzer wissen muß, welche Vektorpaare er einfügen will. Die »Benutzeroberfläche« eines solchen Programms könnte in etwa so aussehen:

```
(A)nhängen, (E)rsetzen, (L)öschen: E
Anzahl der Vektorpaare (dezimal)? 12
1. Vektorpaar (hex): 85 13
2. Vektorpaar (hex): 82 93
   ....
```

Diese Schema ließe sich noch erweitern - etwa in der Form, daß der Benutzer absolute Koordinaten zusammen mit der Information eingibt, ob dabei eine reine Bewegung oder eine Zeichenaktion stattfinden soll und das Programm die sich ergebenden Vektorpaare dann ganz von allein ausrechnet.

»WYSIWYG«

Ein Bearbeitungsprogramm für grafische Objekte beliebiger Art, das die Bezeichnung »Editor« verdient, muß zwei Forderungen erfüllen:

- grafische Darstellung des zu bearbeitenden Objekts;
- Eingabe und Modifikation über einen Grafik-Cursor und in *wahlfreier Reihenfolge.*

Konkret: Das Programm muß in der Lage sein, aus drei aufeinanderfolgenden Linien die mittlere herauszulöschen - und das ist eine ausgesprochen schwierige Angelegenheit, wenn diese Linien als Anweisungs*folge* definiert sind:

```
MoveTo(10,10);
LineTo(20,20);   <== diese Linie soll gelöscht/ersetzt werden
LineTo(20,10);       { Ergebnis: MoveTo(20,20); LineTo(20,10); }
```

Und wo wir gerade dabei sind: Das Programm sollte auch erkennen, daß sich zwei zu verschiedenen Zeitpunkten eingefügte Linien wie

```
MoveTo(10,10);
LineTo(20,20);
  ...
MoveTo(20,20);
LineTo(40,12);
```

mit drei Anweisungen zusammenfassen lassen:

```
MoveTo(10,10);
LineTo(20,20);
LineTo(40,12);
```

Der im folgenden Abschnitt vorgestellte Editor CE löst dieses Problem, indem er

- eine Vektordefinition vor der Bearbeitung in ein Linien-Array mit (absoluten) Anfangs- und Endpunkten umrechnet,
- sich während der Bearbeitung um die Zusammenhänge zwischen einzelnen Linien überhaupt nicht kümmert,
- und die Start- und Endpunkte des Linien-Arrays hinterher wieder in eine Vektorfolge umformt. Diese Umrechnung arbeitet mit einem eigenen »Cursor« und sucht das Array als erstes nach einer Linie ab, die mit der »momentanen Cursorposition« beginnt oder endet. Nur wenn sich keine direkte Fortsetzung zum momentanen Punkt finden läßt, wird ein neuer *MoveTo*-Vektor erzeugt.

Obwohl die entsprechende Routine nicht über mehrere Ebenen vorausdenkt*), scheint sie doch recht effizient zu sein: Mitunter kürzt sie auch aus den Original-Zeichendefinitionen von Borland noch ein oder zwei Vektorpaare heraus.

CE - Der Editor für grafische Zeichensätze

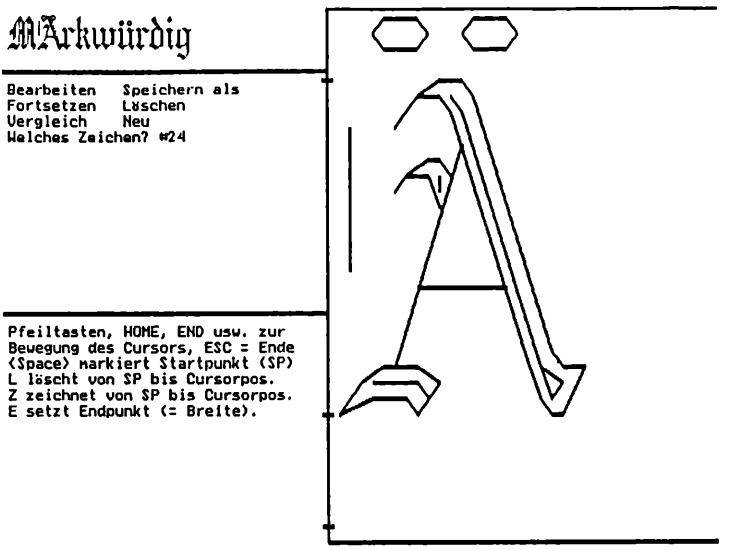
Die Entstehungsgeschichte von CE könnte einem »Programm-Kochbuch« entnommen sein: Nach der Analyse der Prozeduren *OutText* und *SetTextStyle* und der daraus folgenden Feststellung, wie eine .CHR-Datei aufgebaut ist, wurde ein Vorläufer von *DrawChar* geschrieben, der zuerst als Unterprogramm in den Editor und schließlich (nach der Umstellung auf ein Linien-Array) ganz verschwand. Dazu kamen die Routinen zum Anhängen und Löschen von Zeichendefinitionen, eine Dateiverwaltung, die Forderung, daß das Programm größtenteils unabhängig vom verwendeten Video-Adapter läuft, ein Fenster für den Editor, ein Fenster für Menüs, ein weiteres Fenster für Hilfestellungen... und schließlich ist ein Programm daraus geworden, das den Vergleich mit industriellen Standards nicht zu scheuen braucht, dessen Listing aber gut und gerne 20 Seiten füllt.

CE unterteilt den Bildschirm in vier voneinander abgegrenzte Bereiche:

- *Vergleichstext*: Hier läßt sich ein Text mit bis zu 10 Zeichen eingeben, in dem das zu bearbeitende Zeichen einmal vorkommt. Der Text (und das Zeichen) werden nach jeder Veränderung auf den neuesten Stand gebracht - man erhält auf diese Weise einen guten Überblick, wie das »Editor-Zeichen« in einem laufenden Text aussieht, kann nach der Erzeugung eines »ä« sicherstellen, daß sich die Tüpfelchen eines »ö« auf derselben Höhe befinden usw.

*) Anstatt mögliche Wege über mehrere Vektoren hinweg zu verfolgen, nimmt sie einfach jeweils die erste Linie, die eine direkte Fortsetzung vom momentanen Punkt aus darstellt.

- *Menü*: Für die aktuelle (Unter-)Menüzeile und Eingaben. Die Menüauswahl erfolgt durch die Eingabe des jeweiligen Anfangsbuchstabens, die einzelnen Möglichkeiten sind im nächsten Abschnitt beschrieben.
- *Hilfestellung*: Enthält recht detaillierte Beschreibungen und Hinweise zum momentan aktiven Menüpunkt bzw. den Tastenbefehlen des Editors.
- *Editor*: Dieses Fenster nimmt die rechte Hälfte des Bildschirms ein und zeigt das jeweils zu bearbeitende Zeichen in der maximal möglichen Vergrößerung.



CE mit einem Vergleichstext und dem Zeichen »Ä« im Editor

A propos Hilfestellungen: Ich gehe davon aus, daß CE auch von Leuten benutzt wird, die weniger an den programmiertechnischen Details als an Umlauten in ihren Grafik-Zeichensätzen interessiert sind. Deshalb folgt hier eine kurze und »schulmäßige« Beschreibung des Programms:

Der Start von CE

CE wird zusammen mit dem Namen der zu bearbeitenden Zeichensatz-Datei gestartet. Die Angabe des Suffixes .CHR im Namen der Zeichensatz-Datei ist fakultativ, d.h. die Aufrufe `CE goth` und `CE goth.chr` haben dieselbe Wirkung.

In der zunächst vorgestellten Grundversion bearbeitet CE immer nur einen Zeichensatz auf einmal - eine Funktion zum Transfer von Definitionen *zwischen* zwei verschiedenen Zeichensatz-Dateien wird erst in einer weiteren Ausbaustufe definiert.

Die Auswahl von Menüpunkten...

erfolgt über die Eingabe des entsprechenden Anfangsbuchstabens, mit ESC gelangt man jeweils eine Menüebene zurück. Jede Anfrage von CE läßt sich mit ESC abbrechen. Nach der Auswahl von *Zeichen / Löschen* fragt CE beispielsweise mit

```
Welches Zeichen?
```

nach dem zu löschenden Zeichen, das entweder direkt (z.B. »A«) oder mit einem vorangestellten Doppelkreuz und dem darauffolgenden ASCII-Code (z.B. »#65« anstelle von »A«) eingegeben werden kann. Ein Abschluß dieser Eingabe mit RETURN führt die Löschung aus, ein ESC an beliebiger Stelle der Eingabe bricht die Operation folgenlos ab und bringt Sie zum Menü *Zeichen* zurück.

Das Hauptmenü...

ist nach dem Start von CE aktiv und läßt sich von jedem Punkt des Programms aus durch die (ggf. mehrfache) Eingabe von ESC erreichen. Es enthält die folgenden Wahlpunkte:

- *Speichern*: fragt nach dem Namen der Zeichensatz-Datei und speichert den beim Start von CE geladenen Zeichensatz (inklusive eventueller Modifikationen) unter dem angegebenen Dateinamen.
- *Tabelle*: gibt sämtliche definierten Zeichen im Fenster des Editors aus.
- *Zeichen*: führt zu einem weiteren Menü (siehe unten).
- *Ende*: beendet CE. Falls Zeichendefinitionen verändert wurden (und die Datei noch nicht über *Speichern als* auf die Diskette geschrieben ist), erhalten Sie eine entsprechende Rückfrage und die Möglichkeit zur Speicherung. (Der Abbruch dieser Rückfrage mit ESC führt zum Hauptmenü zurück).

Das Untermenü »Zeichen«...

enthält die folgenden Wahlpunkte:

- *Bearbeiten:* fragt nach einem Zeichen, das entweder direkt (z.B. als »A«) oder mit einem Doppelkreuz, gefolgt von einem ASCII-Code (z.B. »#65« anstelle von »A«) eingegeben werden kann, lädt dieses Zeichen in den Arbeitspuffer und aktiviert den Editor.
- *Neu:* löscht den Arbeitspuffer und aktiviert den Editor.
- *Fortsetzen*: aktiviert den Editor, ohne den Arbeitspuffer zu verändern. Falls sich zu diesem Zeitpunkt noch kein Zeichen im Arbeitspuffer befindet (d.h. der Befehl direkt nach dem Start von CE gegeben wurde), wird diese Menüauswahl von CE ignoriert.

- *Vergleich*: fordert die Eingabe eines Vergleichstextes an, der in der oberen linken Ecke des Bildschirms ausgegeben wird. Die Position des »Editor-Zeichens« in diesem Vergleichstext wird mit einem Caret (^) festgelegt:

Vergleichstext	Zeichen im Editor	Ausgabe
T^st	e	Test
^rgerlich	X	Xrgerlich
^rgerlich	Ä	Ärgerlich

CE behandelt nur das erste Vorkommen eines Carets als Platzhalter für das »Editor-Zeichen«:

Vergleichstext	Zeichen im Editor	Ausgabe
T^^st	X	TX^st

- *Löschen:* fragt (wie *Bearbeiten*) nach einem Zeichen und entfernt dessen Definition aus der Zeichensatz-Datei. Nach einer Löschung gilt die Zeichensatz-Datei als verändert (und Sie erhalten eine entsprechende Rückfrage beim Beenden von CE).

- *Speichern als:* führt zu einem weiteren Untermenü mit zwei Wahlpunkten:

  - *Zeichen überschreiben*: fragt (wie *Bearbeiten*) nach einem Zeichen und ersetzt dessen Definition durch den Inhalt des Arbeitspuffers.

  - *Neu anfügen*: fügt der Zeichensatz-Datei den Inhalt des Arbeitspuffers hinzu und gibt den ASCII-Code aus, den diese neue Definition erhalten hat.

  Nach dem Überschreiben oder Anfügen eines Zeichens gilt die Zeichensatz-Datei als verändert (und Sie erhalten eine entsprechende Rückfrage beim Beenden von CE).

Der Editor...

wird vom Untermenü *Zeichen* aus mit *Bearbeiten*, *Fortsetzen* oder *Neu* aktiviert und mit ESC beendet. Eine automatische Speicherung seines Arbeitspuffers in der Zeichensatz-Datei findet *nicht* statt - dazu müssen Sie (nach dem Verlassen des Editors) den Wahlpunkt *Speichern als* des Menüs *Zeichen* benutzen.

Der Editor definiert kein eigenes Flag im Sinne von »Arbeitspuffer-Inhalt verändert, aber noch nicht mit *Speichern als* in die Datei übertragen« - die Auswahl eines anderen Zeichens über *Bearbeiten* und die Funktion *Neu* löschen den Arbeitspuffer deshalb ohne Rückfrage*).

Der Grafik-Cursor des Editors erscheint als blinkendes Kreuzchen und läßt sich mit den Pfeiltasten horizontal bzw. vertikal sowie mit HOME, END, PGUP und PGDN diagonal bewegen. Die erste Eingabe eines Leerzeichens hält die momentane Position des Grafik-Cursors als Startpunkt einer Linie fest und markiert ihn

*) Hier könnte man beispielsweise noch etwas tun.

mit einem weiteren (nicht blinkenden) Kreuzchen, das sich mit der Eingabe eines weiteren Leerzeichens wieder löschen läßt.

Das Zeichnen einer Linie besteht aus zwei Schritten:

- Cursor auf den gewünschten Anfangspunkt setzen und diesen Punkt mit der Eingabe eines Leerzeichens als Startpunkt markieren;
- Cursor auf den gewünschten Endpunkt setzen und Eingabe des Buchstabens »Z«. Diese Operation gibt auch den Startpunkt wieder frei - mehrere miteinander verbundene Linien werden also folgendermaßen gezeichnet:

```
Cursorbewegung <Space> Cursorbewegung Z <Space> Cursorbewegung Z ...
```

Das Löschen einer Linie verläuft im Prinzip genauso: Nach dem Markieren des Anfangspunktes mit dem Cursor und der Leertaste setzen Sie den Cursor auf den Endpunkt der Linie und geben den Befehl *L*.

Der Editor richtet sich sowohl beim Zeichnen als auch beim Löschen nach den Anfangs- und Endpunkten einer Linie und berechnet keine Teilabschnitte. Um eine existierende Linie zu verkürzen, müssen Sie zuerst die gesamte Linie löschen und dann eine neue (kürzere) Linie zeichnen. Dasselbe gilt für das Löschen von »Mittelstücken« einer Linie: Um aus einer Linie wie

```
A -> B -> C -> D
```

das Teilstück B -> C herauszulöschen, müssen Sie die gesamte Linie (A -> D) löschen und danach die beiden Teilstücke A -> B und C -> D neu zeichnen.

Der Editor verhindert automatisch, daß eine Linie mehrfach gezeichnet wird - wenn Sie also im Verlauf einer Bearbeitung zweimal einen Zeichenbefehl A -> D geben, speichert CE diese Linie trotzdem nur einmal. Falls Sie dagegen zuerst eine Linie von A -> D zeichnen, danach eine Linie A -> C usw., dann merkt CE nicht, daß es sich hier teilweise um dieselben*) Linien handelt, und speichert zwei voneinander getrennte Definitionen.

Die Bestimmung der Zeichenbreite...

ist bereits der letzte Befehl des Editors: die Eingabe eines *E* hält die momentane Position des Cursors als Endpunkt der Definition fest. Die vertikale Position dieses Endpunkts sollte mit der Grundlinie zusammenfallen - wenn sie es nicht tut, werden folgende Zeichen bei einer Ausgabe mit *OutText* entsprechend versetzt (womit sich manchmal auch recht hübsche Effekte erzielen lassen).

*) Mit mehrfach überlagerten Teilstücken lassen sich vor allem dann verblüffende Effekte erzielen, wenn man sie später zu löschen versucht - derartige Linien erscheinen dann als ausgesprochen »hartnäckig«.

Die Programmstruktur von CE

CE besteht aus 4 Dateien, von denen 3 als Units formuliert sind:

- *CEGlobal* - enthält die globalen Variablen, Konstanten und Datentypen des Programms;
- *CEUtils* - Routinen zur Umrechnung von Vektordefinitionen in Linien und zurück, dem Einfügen und Löschen von Definitionen, der formatierten Ausgabe von Hilfestellungstexten usw.
- *CEEdit* - der grafische Editor;
- *CE* - das Hauptprogramm.

Das gesamte Programm ist soweit wie irgend möglich im Standard-Pascal gehalten, d.h. arbeitet ohne **inline**-Anweisungen, Stack-Tricks und Seiteneffekte. Zeiger wurden nur an den Stellen eingesetzt, wo es wirklich unumgänglich war.

Das Unit CEGlobal

```
unit CEGlobal;   { CharEdit - 12/12/87 as }

{ Definition globaler Variablen, Konstanten und Datentypen }

Interface

type
  LineType = record
               DnFlag: Boolean;  { "bereits erledigt" }
               Kind: ShortInt;   { LDONE, LMOVE, LDRAW, LDEL }
               Xs,Ys,            { Startpunkt }
               Xe,Ye: Integer;   { Endpunkt }
             end;
```

Der Arbeitspuffer des Editors besteht aus einem *array of LineType* - hier werden die (aus den Vektoren berechneten) Linien einer Zeichendefinition 1:1 gespeichert. Die Vergrößerung (* MAG) wird erst beim tatsächlichen Zeichnen gesetzt.

```
  VecType = record
               Xv, Yv: ShortInt; { X-Vektor, Y-Vektor }
             end;
  VecArray = array[1..200] of VecType;
```

VecType definiert ein Vektorpaar als Record; der Typ *VecArray* macht es möglich, die Startadresse einer Vektordefinition (*@FileBuf[Index]*) zu übergeben und innerhalb der aufgerufenen Routine als Vektor-Array (mit dem Start-Index 1) zu interpretieren.

```
const

  LDONE = 0; LMOVE = 2;  { Linienarten (s. LineType) }
  LDRAW = 3; LDEL = -1;
```

Die drei von *OutText* definierten Funktionscodes und ein vierter Typ, der innerhalb des Linien-Arrays zur Kennzeichnung gelöschter Einträge verwendet wird. Auf diese Weise sparen wir uns das Umsortieren folgender Array-Elemente bei der Löschung einer Linie.

```
var
  CHRFile: File;                    { *.CHR, wird via BlockRead gelesen }
  ChrFName: string;                 { Name der .CHR-Datei }
  FileBuf: array[0..10000] of Byte; { für die .CHR-Datei }
  FLength,                          { tatsächliche Dateilänge }
  FlengthPos: Integer;              { Position innerhalb der .CHR-Datei,
                                     ab der FLength-DATAOffs verzeichnet ist }

  DataOffs: Integer;   { Beginn des "eigentlichen" Dateiinhalts }
  MinChar: Byte;       { ASCII-Code des niedrigsten definierten Zeichens }
  TblSize: Integer;    { Anzahl der Zeichendefinitionen plus 1 }
  TBStart: Integer;    { Start der Vektor-Adresstabelle }
  WidthTbl: Integer;   { Startadresse der Breitentabelle }
  VecStart: Integer;   { Startadresse der ersten Vektordefinition }
  CUp,CDown: ShortInt; { Zeichenhöhe, Unterlänge }
```

Die Variablen zur Speicherung der Eckwerte eines Zeichensatzes sind dieselben wie im Programm *DrawChar* (Seite 348).

```
  Lines: array[1..200] of LineType; { Arbeitspuffer des Editors }
  Vectors: VecArray;                { Zur Rück-Umwandlung einer Zeichen-
                                      definition (Lines) in Vektoren }
```

Die Rück-Umwandlung eines Linien-Arrays in eine Vektorfolge verwendet ein eigenes Vektor-Array zur Zwischenspeicherung. Der »Einbau« von *Vectors* in *FileBuf* (d.h. die Datei) erfolgt erst nach Abschluß der Umwandlung, weil vorher nicht bestimmbar ist, wieviele Vektoren dabei entstehen.

```
  CPStr: string;                    { Vergleichs-String }
  CPStrPtr: Pointer;                { Zeiger auf den 2.Teil davon }
  CP2Start: Integer;                { Startposition 2.Teil CPStr }
  CPStr2Width: Integer;             { Breite 2.Teil CPStr }
  EdCharCX: Integer;                { X-Position EdChar in CPStr }
  Blank: Pointer;                   { ein grafisches Leerzeichen }
  CXStart,CYStart: Integer;         { Startposition 1.Teil CPStr }
```

Weil eine Neuausgabe des gesamten Vergleichstextes nach jeder Veränderung des »Editor-Zeichens« viel zu lange dauern würde, unterteilt das Programm diesen Text in zwei Hälften: »vor dem Editor-Zeichen« und »nach dem Editor-Zeichen« (*CP2Str*). Die erste Hälfte wird statisch ausgegeben, die zweite via *GetImage* in

einem (dynamisch belegten) Bereich gespeichert und nur bei einer Breitenveränderung des Editor-Zeichens mit *PutImage* neu auf den Bildschirm gebracht.

```
WndX, WndY : Integer;              { Ecken des Zeichenfensters }
CsX, CsY: Integer;                 { Cursor-Position im Editor }
Edx1,Edx2,Edy1,Edy2: Integer;      { Ecken des Editorfensters (Pixel) }
EdxMin,EdxMax,
EdyMin,EdyMax: Integer;            { Grenzen der Cursorbewegung (Punkte) }
Liney1, Liney2: Integer;           { Grenzen von CP,Menüs,Hilfstexten }
```

Die Grenzen der vier Bildschirmbereiche (»Fenster«) hängen vom verwendeten Grafiktreiber, dem Video-Adapter und dem Platzbedarf des Zeichensatzes ab - sie werden beim Start des Programms entsprechend gesetzt.

```
EdChar: Char;                  { ASCII-Code des Zeichens im Puffer
                                 des Editors }
MAG  : Integer;                { Vergrößerungsfaktor }
EdCharWidth : Integer;         { Breite von EdChar in Punkten }

GraphDriver, GraphMode: Integer;  { Nun ja ... }

Implementation
                { kein Initialisierungsteil }
end.
```

Das Unit CEUtils

Dieses Unit enthält praktisch alle Routinen, die weder mit dem Hauptprogramm noch mit dem Editor in direkter Verbindung stehen (oder von beiden benutzt werden).

Die Kommentare im Interfaceteil wurden für das Listing im Buch auf ein absolutes Minimum verkürzt, um wenigstens einige Seiten zu sparen.

```
unit CEUtils;     { CharEdit - 18/12/87 as }

Interface

uses Crt,Graph,CEGlobal;

procedure ClearCPWindow;
{ Löscht das Fenster, in dem der Vergleichs-String dargestellt wird }
procedure ClearMenuWindow;
procedure ClearHelpWindow;
procedure ClearEditWindow;
procedure HelpESCtoMenu;  { 'ESC = Zurück zum Hauptmenü' }

procedure GetLine(var s:string; Max:Integer); { grafisches Readln }
procedure OutTextFormat(yp: Integer; Msg: string); { Textausgabe im Fenster }
procedure OutTextError(Msg: string); { Fehlermeldungs-Ausgabe }
```

```
function GetCharIndex(Ch: Char): Integer; { Startadresse von Ch in FileBuf }
procedure VecsToLines(var Source); { Vektoren -> Linien-Array }
function LinesToVecs: Integer;      { Linien-Array -> Vektoren }
function PaintChar(WnX,WnY,Size:Integer; DrawEndP: Boolean): Integer;

procedure DelDefinition(CCode: Byte); { löscht die Definition CCode }
procedure InsertDefinition(CCode: Byte); { fügt die Definition CCode hinzu }

{ ******************************************************* }

Implementation

function GetCharIndex (Ch: Char): Integer;
var x:Integer;
begin
  x := (Ord(Ch) - MinChar) shl 1;
  GetCharIndex := Filebuf[x+TBStart]
              + 256 * FileBuf[x+TBStart+1] + VecStart + DATAOffs;
end;
```

GetCharIndex liefert die Startadresse der Vektorfolge innerhalb von *FileBuf* für das Zeichen in *Ch* zurück. (*MinChar* = ASCII-Code des Zeichens am Tabellenanfang, *TBStart* = Start der Vektor-Adresstabelle, *VecStart* = Startadresse der ersten Vektordefinition, *DATAOffs* = Beginn der »eigentlichen« Daten in *FileBuf*).

```
procedure VecsToLines(var Source);
var
  Done: Boolean;
  CurrX, CurrY: Integer;  { "Cursor" - momentane Zeichenposition }
  XVec, YVec: ShortInt;
  Func: Integer;
  LineIndex, VecIndex: Integer;
begin
  LineIndex := 1; VecIndex := 1; CurrX := 0; CurrY := 0;
  repeat
    with VecArray(Source)[VecIndex] do
    begin
      XVec := Xv; YVec := Yv;
    end;
    Inc(VecIndex);
                        { Funktionscode ermitteln }
    Func := (XVec and $80) shr 6 + (YVec and $80) shr 7;
    XVec := XVec and $7F;
    YVec := YVec and $7F;   { und dann ausblenden  }

    if XVec >= $40  then XVec := -128 + XVec;  { "Vorzeichen" }
    if YVec >= $40 then YVec := -128 + YVec;
```

```
    case Func of
      LDONE: with Lines[LineIndex] do      { Festlegung des Endpunkts }
             begin
               Xs := CurrX; Ys := CurrY;
               Kind := LDONE;
             end;
      LMOVE: begin                         { Bewegung ohne Zeichnen }
               CurrX := XVec; CurrY := YVec;
             end;
      LDRAW: begin                         { Bewegung mit Zeichnen }
               with Lines[LineIndex] do
               begin
                 Xs := CurrX; Ys := CurrY;
                 Xe := XVec; Ye := YVec;
                 Kind := LDRAW;
               end;
              CurrX := XVec; CurrY := YVec;
              Inc(LineIndex);
        end;
    end;  { case }
  until Func = LDONE;
end;
```

VecsToLines wandelt eine Folge von Vektoren in eine Folge von Linien um. *Source* steht entweder für eine Adresse innerhalb von *Filebuf* oder für das Array *Vectors*. Das Ziel ist immer der Arbeitspuffer des Editors.

Das Verfahren ist dasselbe wie bei der Ausgabe eines Zeichens - nur daß hier Anfangs- und Endpunkte im Array *Lines* gespeichert werden. Einen direkten Zähler für die Anzahl der Linien gibt es nicht, das letzte gesetzte Element von *Lines* wird durch den Eintrag LDONE im Feld *Kind* gekennzeichnet (und enthält einen speziellen Wert: den Endpunkt der Definition). Da wir nur zu zeichnende Linien speichern, führt der Funktionscode LMOVE lediglich zu einer Bewegung des »Cursors«.

```
function LinesToVecs: Integer;
var Done: Boolean; { Sämtliche Linien bearbeitet }
    LIndex, VIndex: Integer;  { Indexe in Lines und Vectors }
    CurrX, CurrY: Integer;    { "Cursorposition" }
```

LinesToVecs wandelt die Linienfolge aus dem Arbeitspuffer des Editors (Variable *Lines*) in eine Folge von Vektoren (Variable *Vectors*) um, wobei versucht wird, möglichst viele Linien direkt aufeinanderfolgen zu lassen (d.h. mit möglichst wenig LMOVE-Befehlen auszukommen). Das Funktionsergebnis ist die Anzahl der erzeugten Vektoren.

LinesToVecs verwendet zwei lokale Routinen:

```
function ScanforDraw(Cx,Cy: Integer): Integer;
var x, Temp: Integer;
begin
  x:= 1;
  repeat
    with Lines[x] do                          { Schritt 1: Suche nach einer }
      if (Kind = LDRAW) and not DnFlag then   { Linie, deren Start- oder    }
        begin                                 { Endpunkt mit der momentanen }
          if (Xs = Cx) and (Ys = Cy) then     { Position zusammenfällt }
          begin
            ScanforDraw := x;                 { ... gefunden }
            Exit;
           end
          else if (Xe = Cx) and (Ye = Cy) then
          begin
            Temp := Xe; Xe := Xs; Xs := Temp;  { Umdrehen der Linie }
            Temp := Ye; Ye := Ys; Ys := Temp;
            ScanforDraw := x;
            Exit;
          end;
      end; { if Kind = LDRAW .. }
    Inc(x);
  until Lines[x].Kind = LDONE;

  x := 1;                                     { Schritt 2: Suche nach einer }
  repeat                                      { x-beliebigen Linie, die noch }
    with Lines[x] do                          { nicht verzeichnet worden ist }
      if (Kind = LDRAW) and not DnFlag then
      begin
        ScanforDraw := x;                     { ... gefunden }
        Exit;
      end;
    Inc(x);
  until Lines[x].Kind = LDONE;
  ScanforDraw := 0;                           { keine weiteren Linien mehr }
end;  {ScanforDraw}
```

Die zu *LinesToVecs* lokale Funktion *ScanforDraw* sucht das Array *Lines* zuerst nach einem LDRAW-Befehl ab, dessen Startpunkt mit den Koordinaten *Cx,Cy* (der momentanen »Cursorposition«) zusammenfällt. Falls sich kein derartiger Befehl findet, wird der nächste »noch nicht erledigte« LDRAW-Befehl verwendet. Das Funktionsergebnis ist der Index zum gefundenen Befehl bzw. der Wert 0, wenn bereits alle LDRAW-Befehle codiert worden sind.

```
procedure SetVec(Xp,Yp,Func: Integer);
begin
  Inc(VIndex);                    { Index innerhalb von Vectors }
  if Xp < 0 then Xp := Xp + 128;  { Umcodierung für negative Werte }
  if Yp < 0 then Yp := Yp + 128;
  with Vectors[VIndex] do
  begin
    Byte(Xv) := Xp or (Func and $02) shl 6;  { Einbau der Befehlsart (LDONE, }
    Byte(Yv) := Yp or (Func and $01) shl 7;  { LMOVE, oder LDRAW) }
  end;
end;  {SetVec}
```

Die Prozedur *SetVec* ist ebenfalls lokal zu *LinesToVecs* und erzeugt mit den übergebenen Parametern einen Vektor im Array *Vectors*. Der Index in *Vectors* (der letztendlich das Funktionsergebnis von *LinesToVecs* darstellt), wird hier vor der Speicherung erhöht.

```
begin { LinestoVecs }
   LIndex := 1;
   while Lines[LIndex].Kind <> LDONE do   { alle Linien-Records als }
   begin                                  { noch nicht erledigt markieren }
     Lines[LIndex].DnFlag := False;
     Inc(LIndex);
   end;

   VIndex := 0;   { Index in Vectors }
   CurrX:= $80; CurrY := $80; { mom. Position  = "ungültig" }
   repeat
     LIndex := ScanforDraw(CurrX,CurrY);
     if LIndex <> 0 then                  { LDRAW-Befehl gefunden }
     begin
       with Lines[LIndex] do
       begin
         if (Xs <> CurrX) or (Ys <> CurrY) then   { ... keine direkte Fort- }
            SetVec(Xs,Ys,LMOVE);                  { setzung: LMOVE-Befehl }
         SetVec(Xe,Ye,LDRAW);              { danach der LDRAW-Befehl }
         CurrX := Xe; CurrY := Ye;         { neuer Endpunkt }
         DnFlag := True;                   { diese Linie ist erledigt }
       end; { with Lines }
     end; { if LIndex }
   until LIndex = 0;

   LIndex := 1;                            { Suche nach dem LDONE-Befehl }
   while(Lines[LIndex].Kind <> LDONE) do Inc(LIndex);
   with Lines[LIndex] do SetVec(Xs,Ys,LMOVE);    { MOVE zum Endpunkt }
   SetVec(0,0,LDONE);
   LinesToVecs := VIndex;
end;  { LinesToVecs }
```

Der Rumpf von *LinesToVecs*: Zuerst werden sämtliche in *Lines* gespeicherten Linien als »noch nicht erledigt« markiert, danach sucht die Routine über *ScanforDraw* das Array nach zueinander passenden Anfangs- und Endpunkten ab. Wenn die Suche über *ScanforDraw* erfolglos bleibt, wird für die jeweils nächste noch nicht erledigte Linie ein separater LMOVE-Befehl eingefügt.

Nachdem sämtliche Zeichenbefehle in Vektoren umgewandelt worden sind, sucht die Routine über die zweite Schleife nach dem Endpunkt der Definition, der von *VecsToLines* in einem Linien-Element mit dem Funktionstyp LDONE gespeichert wurde. Weil es nach dem Aneinanderhängen der einzelnen Linien nicht vorhersagbar ist, wo sich der Cursor schließlich befindet, wird die Bewegung zum Endpunkt grundsätzlich als separater LMOVE-Befehl codiert*). Ein 00 00-Vektor schließt die Definition ab.

```
function PaintChar(WnX,WnY,Size:Integer; DrawEndP: Boolean): Integer;
var
  x: Integer;  { Index in Lines }
begin
  x:= 1;
  while Lines[x].Kind <> LDONE do   { von der 1. bis zur letzten Linie }
  begin
    with Lines[x] do
      if Kind <> LDEL then   { ... nicht gelöscht }
        Line(WnX+Xs*Size,WnY-Ys*Size,WnX+Xe*Size,WnY-Ye*Size);  { Zeichnen }
    Inc(x);
  end;
  if DrawEndP then with Lines[x] do   { Endpunkt zeichnen? }
    PutPixel(WnX+Xs*Size,WnY+Ys*Size,GetColor);
  PaintChar := Lines[x].Xs;       { Breite des Zeichens }
end;
```

PaintChar ist die eigentliche Zeichenroutine des Editors und bringt die in *Lines* definierten Linien im durch *WnX* und *WnY* festgelegten »Fenster« mit dem Vergrößerungsfaktor *Size* auf den Bildschirm. Das Funktionsergebnis ist die Breite des Zeichens (d.h. die X-Koordinate des Endpunkts).

```
procedure ClearCPWindow;
{ Löscht das Fenster, in dem der Vergleichs-String dargestellt wird }
begin
  SetViewPort(0,0,Edx1-6,Liney1-5,True);
  ClearViewPort;
  SetViewPort(0,0,GetMaxX,GetMaxY,True);
end;
```

*) Hier hätte man allerdings wenigstens noch prüfen können, ob sich der Cursor zufällig auf dem gewünschten Endpunkt befindet - so unwahrscheinlich das auch ist.

```
procedure ClearMenuWindow;
{ Löscht das Fenster, in dem die Menüs und Anfragen ausgegeben werden }
begin
  SetViewPort(0,Liney1+5,Edx1-6,Liney2-5,True);
  ClearViewPort;  SetViewPort(0,0,GetMaxX,GetMaxY,True);
end;

procedure ClearHelpWindow;
{ Löscht das Fenster, in dem die Texte der Hilfestellung ausgegeben werden }
begin
  SetViewPort(0,Liney2+5,Edx1-6,GetMaxY,True);
  ClearViewPort;  SetViewPort(0,0,GetMaxX,GetMaxY,True);
end;

procedure ClearEditWindow;
{ Löscht das Fenster des Editors }
begin
  SetViewPort(Edx1+MAG-1,Edy1+MAG-1,Edx2,Edy2-MAG,True);
  ClearViewPort;
  SetViewPort(0,0,GetMaxX,GetMaxY,True);
end;
```

Die vier Routinen zum Löschen der Fenster von CE.

```
procedure GetLine(var s:string; Max:Integer);
var Ch: Char;
begin
  s:= '';
  repeat
    Ch := ReadKey;
    if (Ch >= ' ') and (Byte(s[0]) < Max) then
    begin
      s:= s + Ch; OutText(Ch);
    end
    else if (Ch = #8) or ((Ch = #0) and (ReadKey = #75)) then  {** $B+! **}
      if s[0] > #0 then   { Backspace, Linkspfeil }
      begin
        MoveRel(-8,0);
        PutImage(GetX,GetY,Blank^,NormalPut);  { <- grafisches Leerzeichen }
        Dec (Byte(s[0]));  { String verkürzen }
      end;
  until (Ch = #13) or (Ch = #27);  { Loop bis CR oder ESC }
  if Ch = #27 then s[0] := #0;   { ESC -> ein Nullstring als Ergebnis }
end;
```

GetLine ist eine Art grafisches *Readln*, arbeitet ab der momentanen Position des Grafik-Cursors und liefert beim Abbruch mit ESC einen Nullstring als Ergebnis. Die Routine reagiert nur auf BS, CR und ESC als Steuerzeichen und funktioniert

ausschließlich mit dem Standard-Zeichensatz. Der Wert von *Max* begrenzt die Länge der Eingabe.

Weil *OutText* zur Zeichenausgabe eine OR-Funktion verwendet, arbeitet die Löschung eines Zeichens (BS) mit *PutImage* und einem grafischen Leerzeichen (*Blank^*), das irgendwann zu Beginn des Programms erzeugt und mit *GetImage* »eingefangen« wird.

```
procedure HelpESCtoMenu;
begin
  ClearHelpWindow;
  OutTextXY(5,Liney2+8, 'ESC = Zurück zum Hauptmenü');
end;
```

Dieses Programmstückchen wird so oft innerhalb der Hauptschleife von CE benötigt, daß es schließlich als eigene Prozedur in *CEUtils* gelandet ist.

```
procedure OutTextFormat(yp: Integer; Msg: string);
const WndWidth = 34;      { Breite des Fensters }
      XStart = 5;    { horizontale Einrückung }
var Start,x,CRPos:Integer;
    Line: string;
begin
   Start := 1;
   repeat
     Line := Copy(Msg,Start,WndWidth);     { maximal 34 Zeichen }
     CRPos:= Pos(#13,Line);
     if CRPos > 0 then        { Zeilenvorschub im String: }
     begin
       OutTextXY(XStart,yp,Copy(Line,1,CRPos-1));  { Ausgabe }
       Inc(Start,CRPos);  { und Index über das CR hinaus }
     end
     else begin
      x:= Length(Line);  { Text kürzer als Fensterbreite? }
      if x < WndWidth then OutTextXY(XStart,yp,Line) { ja, Ausgabe }
        else begin        { sonst Suche nach ' ' }
          while (Line[x] <> ' ') do Dec(x);  { ohne Sicherung! }
          OutTextXY(XStart,yp,Copy(Line,1,x-1)); { und Ausgabe }
        end;
      Inc(Start,x);  { Index + Anzahl ausgegebene Zeichen }
     end;
     Inc(yp,10);      { nächste Zeile }
   until Line = '';  { bis keine weiteren Zeichen mehr übrig sind }
end;
```

OutTextFormat gibt den als *Msg* übergebenen String in einem Fenster mit 34 Zeichen Breite (= *WndWidth*) ab der Vertikalposition *yp* aus und nimmt dabei einen automatischen Zeilenumbruch vor. In *Msg* enthaltene Zeilenvorschübe werden erkannt und korrekt behandelt. CE verwendet diese Routine für die Ausgabe von Menüs und Texten der Hilfestellung.

Vorsicht: *OutTextFormat* erkennt nur Leerzeichen als Trennmöglichkeit und enthält keine Sicherung gegen »untrennbare« Wörter - die Schleife läuft endlos, wenn *Msg* ein durchgehendes Wort mit mehr als 34 Zeichen enthält.

```
procedure OutTextError(Msg: string);
var ch: Char;
begin
  Msg:= Msg + 'Weiter mit einem beliebigen Tastendruck.';
  ClearHelpWindow;
  OutTextFormat(Liney2+8,Msg);
  while KeyPressed do Ch:= ReadKey;  { Tastaturpuffer leeren }
  Ch:= ReadKey;
  while KeyPressed do Ch := ReadKey; { falls da jemand auf eine Funktions- }
end;                                 { taste gedrückt hat ... }
```

OutTextError gibt die als *Msg* übergebene Fehlermeldung via *OutTextFormat* im Hilfstext-Fenster aus und wartet dann auf einen Tastendruck.

```
procedure SetCHRFileParms;
begin
  FileBuf[DATAOffs+1] := TblSize;  { Länge der Tabelle + 1 }
  FileBuf[DATAOffs+4] := MinChar;
  FileBuf[DATAOffs+5] := Lo(VecStart); FileBuf[DATAOffs+6] := Hi(VecStart);
  FileBuf[FLengthPos] := Lo(FLength-DATAOffs);
  FileBuf[FLengthPos+1] := Hi(FLength-DATAOffs);
end;
```

SetCHRFileParms wird nach dem Anhängen oder Löschen einer Zeichendefinition aufgerufen und bringt die Eckwerte der Datei auf den jeweils neuesten Stand.

```
procedure SetRelAddrs(ChangeAddr, Offset: Integer);
var x, RelAddr: Integer;
begin
  x:= TbStart;
  while x < WidthTbl do   { bis zum Anfang der Breitentabelle }
  begin
    RelAddr := (FileBuf[x] + FileBuf[x+1] shl 8);  { 16-Bit-Wort }
    if RelAddr > ChangeAddr then
    begin
       RelAddr := RelAddr + Offset;
       FileBuf[x] := Lo(RelAddr); FileBuf[x+1] := Hi(RelAddr);
     end;
    Inc(x,2);
  end;
end;
```

SetRelAddrs wird nach dem Löschen oder Einfügen einer Definition aufgerufen und bringt die Einträge der Vektor-Adresstabelle auf den neuesten Stand, wobei es hier nicht um die ASCII-Codes, sondern um die Adressen eventuell verschobener Vektordefinitionen geht: die gesamte Tabelle wird nach Einträgen abgesucht, die

auf höhere Adressen als *ChangeAddr* (dem Start der veränderten Definition) zeigen. Je nachdem, ob diese Definition verkürzt oder verlängert wurde, hat *Offset* einen negativen oder positiven Wert.

```
procedure DelDefinition(CCode: Byte);
{ Löscht das Zeichen mit dem ASCII-Code CCode aus dem Zeichensatz. }
var x,Count,RelAddr,DefAddr,DefIndex: Integer;
    DoubleDef: Boolean;

begin
  DefIndex := (CCode - MinChar) shl 1; { Position in der Tabelle }
  RelAddr := FileBuf[DefIndex+TBStart]     { Relativzeiger zur Definition }
             + FileBuf[DefIndex+TBStart+1] shl 8;
  DefAddr := RelAddr + VecStart + DATAOffs; { Abs. Adresse der Definition }

  { Schritt 1: Prüfung, ob diese Definition von mehreren Zeichen
               verwendet wird. }
  Count := 0;
  for x:= 0 to 127-MinChar do
    if FileBuf[x shl 1 + TBStart] +
      Filebuf[x shl 1 + TBStart+1] shl 8 = RelAddr
        then Inc(Count);
  DoubleDef := (Count > 1);  { True, wenn mehr als ein Relativzeiger
                               auf diese Vektordefinition zeigt }
```

Der erste Schritt bei der Löschung einer Definition: Wenn mehr als ein Eintrag der Vektor-Adresstabelle auf die Vektorfolge zeigt, dürfen die Vektoren nicht gelöscht bzw. verkürzt werden. Stattdessen müssen wir ein zusätzliches 00 00-Vektorpaar ans Dateiende anhängen und den zu löschenden Eintrag in der Vektor-Adresstabelle auf dieses Vektorpaar setzen - also die Datei *verlängern*.

```
  { 2. Schritt: Entfernung "einfacher" Definitionen }
  if not DoubleDef then
  begin
    Count := 0;                          { Suche nach dem LDONE-Vektor }
    while FileBuf[DefAddr+Count] or      { der Definition. Dieser Vektor }
      FileBuf[DefAddr+Count+1] > 127 do  { soll als einziges übrigbleiben }
        Inc(Count,2);
    if CCode = MinChar then      { ... können wir auch den LDONE-Vektor }
        Inc(Count,2);            { entfernen, weil der Tabelleneintrag }
                                 { selbst gelöscht wird. }

    Dec(FLength,Count); { Datei entsprechend verkürzen }
    Move(FileBuf[DefAddr+Count],FileBuf[DefAddr],FLength-DefAddr);
    FileBuf[WidthTbl+CCode-MinChar] := 0; { Zeichenbreite auf 0 }

    { Anpassung der Relativzeiger für die nachfolgenden Definitionen }
      SetRelAddrs(RelAddr,-Count);
  end
```

Der zweite Schritt: Wenn die Vektordefinition nicht mehrfach benutzt wird, dann können wir sie bis auf den abschließenden LDONE-Vektor aus der Datei entfernen. Falls das Zeichen als allererstes in der Vektor-Adresstabelle steht, wird die Tabelle in Schritt 3 entsprechend verkürzt - wir können in diesem Fall also auch den LDONE-Vektor hinauswerfen.

```
else if CCode <> MinChar then { Die Vektordefinition wird mehrfach }
begin                         { verwendet. Erzeugung eines neuen   }
  FileBuf[FLength] := 0;      { LDONE-Vektors. }
  FileBuf[FLength+1] := 0;
  RelAddr := FLength - VecStart - DATAOffs;
  FileBuf[DefIndex+TBStart] := Lo(RelAddr);   { Eintrag der neuen }
  FileBuf[DefIndex+TBStart+1] := Hi(RelAddr); { Relativadresse }
  Inc(FLength,2);             { Dateilänge plus LDONE-Vektor (2 Bytes) }

  { Eintrag in der Breitentabelle auf 0. Hier steht für jedes Zeichen
    ein eigener Eintrag, wir müssen also Gott sei Dank nicht erweitern }
  FileBuf[WidthTbl+CCode-MinChar] := 0;
end;
```

Für mehrfach verwendete Vektordefinitionen (*DoubleDef* = *True*) wird hier das bereits erwähnte 00 00-Vektorpaar ans Dateiende angehängt - es sei denn, das zu löschende Zeichen ist das erste in der Tabelle (*MinChar*). In diesem Fall ist die Erzeugung eines 00 00-Vektorpaars nicht notwendig, weil im nächsten Schritt der Tabelleneintrag selbst entfernt wird.

```
{ 3. Tabellenverkürzung für CCode = MinChar }
if CCode = MinChar then
begin
  Dec(FLength,2);                 { Datei um zwei Bytes verkürzen }
  Dec(VecStart,2);                { Vektoren-Startadresse - 2 }
  Inc(MinChar);                   { MinChar um eins nach oben }
  Move(FileBuf[TBStart+2],        { Vektor-Adresstabelle um 2 nach vorne }
       FileBuf[TBStart],FLength-TBStart);
  Dec(TblSize);                   { Gesamtzahl der Definitionen -1 }
  Dec(WidthTbl,2);                { Breitentabelle um 2 nach vorne }
  Dec(FLength);                   { Datei um ein weiteres Byte verkürzen }
  Move(FileBuf[WidthTbl+1],       { Breitentabelle & Vektoren }
       FileBuf[WidthTbl],FLength-WidthTbl); { um 1 nach vorne }
  Dec(VecStart);                  { Tabelle um 1 verkürzt }
end;
```

Wenn das zu löschende Zeichen das allererste in der Vektor-Adresstabelle ist, dann wurde die Vektordefinition zuvor entweder komplett gelöscht oder (für *DoubleDef* = *True*) kein eigener LDONE-Vektor erzeugt. Die Vektor-Adresstabelle wird um zwei Bytes nach vorne verschoben (und der allererste Eintrag damit überschrieben). Die Breitenangabe für dieses Zeichen muß danach ebenfalls gelöscht werden,

damit der Zusammenhang zwischen ASCII-Codes und Einträgen wieder stimmt - folglich erniedrigt sich die Startadresse sämtlicher Vektoren um ein weiteres Byte.

```
    SetCHRFileParms;    { Eckwerte neu setzen }
  end; { DelDefinition }
```

Zu guter Letzt werden die in der .CHR-Datei direkt eingetragenen Parameter über den Aufruf von *SetCHRFileParms* auf den neuesten Stand gebracht.

```
procedure InsertDefinition(CCode: Byte);
{ Fügt dem Zeichensatz ein Zeichen als "ASCII-Code" CCode hinzu.}
var AddBytes, NumVecs: Integer;
    DefIndex, RelAddr, DefAddr: Integer;
begin
   NumVecs := LinesToVecs; { Umwandlung Linien -> Vektoren }
```

Der erste Schritt beim Einfügen/Anhängen einer neuen Zeichendefinition: Das Zeichen ist im Array *Lines* als Linienfolge gespeichert und muß zuerst in eine Vektorfolge umgewandelt werden.

```
   if CCode < MinChar then { neues Zeichen zu Anfang der Tabelle einfügen }
   begin                   { Tabelle um zwei Bytes nach hinten }
     Move(FileBuf[TbStart],FileBuf[TBStart+2],FLength-TbStart);
     Inc(FLength,2);
     Dec(MinChar); Inc(TblSize); { Tabelle ist nach unten erweitert }
     Inc(VecStart,2); Inc(WidthTbl,2);

     { Eintrag des Relativzeigers }
     FileBuf[TbStart] := Lo(FLength-VecStart-DATAOffs);
     FileBuf[TbStart+1] := Hi(FLength-VecStart-DATAOffs);

     { Vektordefinition dazukopieren }
     Move(Vectors,FileBuf[FLength], NumVecs*2);
     Inc(FLength,NumVecs*2);    { und Dateilänge entsprechend erhöhen }

     { Die Tabelle ist um ein weiteres Byte verlängert: }
     Inc(VecStart);    { Vektor-Startadresse erhöhen und Vektoren verschieben }
     Move(FileBuf[WidthTbl],FileBuf[WidthTbl+1],FLength-WidthTbl);
     Inc(FLength);     { und "Verlängerung" der Datei }

     FileBuf[WidthTbl] := EdCharWidth;  { Eintrag der Breite des Zeichens }
   end
```

Das Anfügen eines neuen Zeichens: Wenn der Code dieses Zeichens unterhalb von *MinChar* liegt, wird die Vektor-Adresstabelle nach unten hin erweitert. (Der tatsächliche Wert von *CCode* spielt hier keine Rolle, weil sich durch diese Verlängerung automatisch *MinChar*-1 ergibt). Die Vektor-Adresstabelle wird durch den neuen Eintrag um zwei, die Zeichenbreiten-Tabelle um ein Byte verlängert - insgesamt verschieben sich die Vektordefinitionen also um drei Bytes nach hinten.

```
    else begin
      DefIndex := (CCode - MinChar) shl 1; { Position in der Tabelle }
      RelAddr := FileBuf[DefIndex+TbStart] + FileBuf[DefIndex+TbStart+1] shl 8;
      DefAddr := RelAddr+VecStart+DATAOffs;    { Startadresse der Definition }
      AddBytes := NumVecs*2-2;  { Der LDONE-Vektor existiert hier bereits }
      Move(FileBuf[DefAddr],    { Platz schaffen für die neuen Vektoren }
           FileBuf[DefAddr+AddBytes],FLength-DefAddr);
      Move(Vectors,FileBuf[DefAddr],NumVecs*2); { und Einkopieren der Vektoren,
                                                  (inklusive LDONE-Vektor) }
      Inc(FLength,AddBytes);  { Dateilänge entsprechend erhöhen }
      SetRelAddrs(RelAddr,AddBytes);  { nachfolgende Relativzeiger erhöhen }
      FileBuf[WidthTbl+CCode-MinChar] := EdCharWidth;  { Breite eintragen }
    end;
```

Das Überschreiben einer existierenden Definition funktioniert in dieser Form nur dann, wenn die Vektoren dieser Definition zuvor durch einen Aufruf von *DelDefinition* auf ein einziges 00 00-Vektorpaar reduziert worden sind. *InsertDefinition* verschiebt den Rest der Datei nach hinten, fügt die neue Definition (ohne den abschließenden LDONE-Vektor) ein und korrigiert danach die Einträge der Vektor-Adresstabelle über einen Aufruf von *SetRelAddrs*, deren Vektordefinitionen durch die Erweiterung verschoben worden sind.

```
    SetCHRFileParms;  { Eintrag von FLength, MinChar usw. }
  end;  { InsertDefinition }
```

Der letzte Schritt von *InsertDefinition* ist für beide Modi (Anhängen und Überschreiben) derselbe: Die direkt in der Datei gespeicherten Parameter werden auf den neuesten Stand gebracht.

```
  { ******************************************************** }

        { kein Initialisierungsteil }
  end.
```

Das Unit CEEdit...

definiert nur drei »öffentliche« Prozeduren: die Ausgabe von Zeichen direkt über die Vektordefinition (für den Vergleichstext und den Menüpunkt »Tabelle«), die Eingabe des Vergleichstextes und schließlich den Editor selbst.

```
  unit CEEdit;      { CharEdit - 19/12/87, 13/12/88 as }

  Interface

  uses Crt,Graph,CEGlobal,CEUtils;

  procedure PaintfromVecs (var Source; CxPoint,CyPoint: Integer);
  procedure GetCPStr;   { Eingabe eines Strings zum Vergleich }
  procedure Editor;
  { ********************************************************** }
```

```
Implementation
type
  CursorArray = array[0..4] of ShortInt;  { Zur Cursordefinition und
                                            Hintergrund-Speicherung }
const
  CursorDots: CursorArray = (Cyan,Cyan,Cyan,Cyan,Cyan); { Der Cursor }
  XCoords: CursorArray = (0,-1,0,1,0);    { Relativwerte zur momentanen }
  YCoords: CursorArray = (0,0,-1,0,1);    { Cursorpos. (s. SetCursor) }
  TICKTIME = 6;   { Blinkfrequenz des Cursors in 18,23/Sekunden }

var
  SpX,SpY: Integer;  { Startpunkt-Position }
  Ch: Char;
  EdDone,SpSet, CursorOn: Boolean;
  SysTicks : LongInt absolute $40:$6C;  { "Clock Ticks" }
  NextTick: LongInt;     { Zeitpunkt des nächsten GC-Wechsels }
  CursorSave,SpSave: CursorArray;  { Speicherung Hintergrund 1, 2 }
```

Der vom Editor verwendete Cursor hat die Form eines Pluszeichens und besteht aus fünf Punkten, deren relative Koordinaten hier der Einfachheit halber als Array formuliert sind. Da wir nicht mit einer XOR-Funktion arbeiten können, weil sich der fixe und der blinkende Cursor beim Markieren eines Startpunkts auf derselben Position befinden*), wird der jeweilige Hintergrund in den Arrays *CursorSave* und *SpSave* gespeichert.

Der stetige Wechsel zwischen »Cursor an« und »Cursor aus« - sprich: der Blinkeffekt - wird hier maschinenunabhängig über die »BIOS-Ticks« des Systems zusammen mit der Konstanten TICKTIME festgelegt (vgl. »Benchmarks« in Kapitel 2).

```
procedure PaintfromVecs (var Source; CxPoint,CyPoint: Integer);
var  VecIndex,XVec,YVec, Func: Integer;
begin
  VecIndex:= 1;
  repeat
    with VecArray(Source)[VecIndex] do
    begin
      XVec := Xv; YVec := Yv;
    end;
    Inc(VecIndex);
    Func := (XVec and $80) shr 6 + (YVec and $80) shr 7; { Funktion }
    XVec := XVec and $7F; YVec := YVec and $7F;  { FCode raus }
    if XVec >= $40  then XVec := -128 + XVec;   { Vorzeichen }
    if YVec >= $40  then YVec := -128 + YVec;
```

*) Ehrlich gesagt: Mir sind die Komplikationen bei einer XOR-Funktion über den Kopf gewachsen. Irgendwie müßte das eigentlich schon gehen...

```
      case Func of
        LMOVE: MoveTo(CxPoint+XVec,CyPoint-YVec);  { Bewegung ohne Zeichnen }
        LDRAW: LineTo(CxPoint+XVec,CyPoint-YVec);  { Bewegung mit Zeichnen }
      end;  { case }
    until Func = LDONE;
  end;
```

PaintfromVecs wird für die Ausgabe des Vergleichstextes und von der Funktion »Tabelle« mit der Adresse der jeweiligen Vektordefinition (*@FileBuf[StartIndex]*) aufgerufen und interpretiert diese Adresse als Start eines Arrays mit Vektorpaaren (vgl. Typdefinition in *CEGlobals*). Die Zeichendefinition wird ab dem Punkt *CxPoint,CyPoint* mit dem Vergrößerungsfaktor 1 ausgegeben, der Grafik-Cursor bleibt nach der Ausgabe auf dem Endpunkt des Zeichens stehen.

```
procedure GetCPStr; { Eingabe eines neuen Vergleichs-Strings }
var CXpos,CYpos: Integer;
    TempStr: string;

procedure WriteCPStr(s:string);  { Gibt einen grafischen String aus }
var x:Integer;
begin
  for x:= 1 to Length(s) do
  begin
    PaintfromVecs(FileBuf[GetCharIndex(s[x])],CXpos,CYpos);
    CXpos := GetX;
  end;
end;  { WriteCPStr }
```

Die Ausgabe des Vergleichstextes ist lokal zu *GetCPStr* - über sie wird zuerst der »linke« Teil des Vergleichstextes (d.h. vor dem Editor-Zeichen) und später der »rechte« Teil dieses Strings ausgegeben.

```
procedure MakeControls;
var TVal,x,VLength,Code: Integer;
begin
  x:= 1;
  while x < Ord(TempStr[0]) do  { bis zum vorletzten Zeichen }
  begin
    if (TempStr[x] = '#') then
    begin  { Schlichtes Ausprobieren: Ergeben die auf '#' folgenden }
           { Zeichen einen numerischen Wert? }
      VLength := 3;
      repeat
        Val(Copy(TempStr,x+1,VLength),TVal,Code); { Numerisch? }
        if Code <> 0 then Dec(VLength);
      until (VLength = 0) or (Code = 0);
```

```
      if VLength > 0 then
      begin     { ... ja: das Doppelkreuz wird durch das entsprechende }
                { Zeichen ersetzt, die Ziffern danach herausgelöscht }
        TempStr[x] := Chr(Lo(TVal));
        Delete(TempStr,x+1,VLength);
       end;
    end;
  Inc(x);
  end;
```

MakeControls **ist nur aus Gründen der Übersichtlichkeit als eigene Routine deklariert. Im ersten Teil dieser Prozedur werden alle Vorkommen der Zeichenfolge *#nnn* daraufhin geprüft, ob sich *nnn* numerisch auswerten läßt - wenn ja, wird *#nnn* (inklusive dem Doppelkreuz) durch ein einzelnes Zeichen mit dem ASCII-Code *nnn* ersetzt.**

```
      { 2. Filterung: Alle Zeichen, die außerhalb der Grenzen liegen,
        werden aus dem String entfernt }
  x:= 1;
  while x <= Length(TempStr) do
  begin
   if (TempStr[x] < Chr(MinChar)) or (TempStr[x] > Chr(MinChar+TblSize-1))
     then Delete(TempStr,x,1)   { .. raus mit dir! }
     else Inc(x);  { (nach Delete zeigt x bereits auf das nächste Zeichen) }
  end;
end;  { MakeControls }
```

Die zweite Filterung in *MakeControls* wirft alle Zeichen aus dem String hinaus, die sich nicht mit dem geladenen Zeichensatz darstellen lassen.

```
begin { GetCPStr }
  GetLine(TempStr,15);                  { Eingabe }
  if TempStr = '' then Exit;            { -> Abbruch von GetLine mit ESC }
  MakeControls;                         { #xxx -> Steuerzeichen }
  if CPStrPtr <> nil then     { Freigabe des alten Speicherbereichs }
  FreeMem(CPStrPtr, ImageSize(0,0,CPStr2Width,CUp+3-CDown));
  CPStr:= Copy(TempStr,1,10);           { Begrenzung auf 10 Zeichen }
  ClearCPWindow;
  CXpos := CXStart; CYpos := CYStart;
  WriteCPStr(Copy(CPStr,1,Pos('^',CPStr)-1));
  EdCharCX := CXpos;                    { Breite des ersten Teils }
  CP2Start := EdCharCX + EdCharWidth; { Start des zweiten Teils }
  CXpos := CP2Start;
  WriteCPStr(Copy(CPStr,Pos('^',CPStr)+1,255));
  CPStr2Width := CXpos-CP2Start;        { Breite des zweiten Teils }
  GetMem(CPStrPtr,ImageSize(0,0,CPStr2Width,CUp+3-CDown));
  GetImage(CP2Start,CYStart-3-CUp,CXpos,CYStart-CDown,CPStrPtr^);
end;
```

Der Rumpf von *GetCPStr*: Eingabe des Vergleichstextes durch den Benutzer via *GetLine*, »Filterung« via *MakeControls* und Ausgabe des ersten Teils (bis zum »^«, dem Platzhalter für das Editor-Zeichen).

Die Horizontalposition nach der Ausgabe des ersten Teils wird als Startpunkt für das Zeichnen des Editor-Zeichens in der Variablen *EdCharCX* festgehalten - ab dieser Position zeichnet der Editor »sein« Zeichen. Die Addition der Breite des Editor-Zeichens (*EdCharWidth*) ergibt den Startpunkt für den zweiten Teil des Vergleichstextes, der danach ausgegeben und via *GetImage* gespeichert wird.

Für den Fall, daß der Benutzer die Breite des Editor-Zeichens zu einem späteren Zeitpunkt verändert (oder ein anderes Zeichen in den Editor lädt), läßt sich deshalb der zweite Teil des Vergleichstextes einfach an der entsprechenden Stelle (*EdCharCX* + *EdCharWidth*) mit *PutImage* neu ausgeben.

```
procedure SaveCArray(XP,YP: Integer; var Target: CursorArray);
var x: Integer;
begin
  Xp := WndX+XP*MAG; yp := WndY-YP*MAG;
  for x:= 0 to 4 do
    Target[x] := GetPixel(Xp+XCoords[x],Yp+YCoords[x]);
end;
```

SaveCArray speichert fünf Pixel des Bildes an der Position *XP,YP* in einem Array, d.h. rettet den Hintergrund, bevor einer der beiden Editor-Cursoren gezeichnet wird. Die Ziel-Variable *Target* ist entweder das Array *CursorSave* (für den blinkenden Cursor) oder das Array *SPSave* (bei der Markierung eines Startpunkts).

Cursorpositionen werden nicht mit absoluten Koordinaten, sondern in derselben Weise wie die Liniendefinitionen behandelt (d.h. relativ zu den Fenstergrenzen des Editors und unabhängig vom Vergrößerungsfaktor MAG).

```
procedure SetCursor(XP,YP: Integer; GC: CursorArray);
var x: Integer;
begin
  Xp := WndX+XP*MAG; yp := WndY-YP*MAG;
  for x:= 0 to 4 do
    PutPixel(Xp+XCoords[x],Yp+YCoords[x],GC[x]);
end;
```

Der Name *SetCursor* für diese Routine ist etwas irreführend - je nachdem, welches der Arrays hier als *GC* übergeben wird, zeichnet sie entweder einen Cursor oder stellt einen Hintergrund wieder her.

```
procedure GCursor;
begin
  if NextTick > SysTicks then Exit;  { Zeit noch nicht abgelaufen }

  NextTick := SysTicks + TICKTIME;  { neuer Umschaltpunkt "in der Zukunft" }
  CursorOn := not CursorOn;  { Umschaltung }
```

```
    if CursorOn then
    begin
       SaveCArray(CsX,CsY,CursorSave);  { Bildinhalt speichern }
       SetCursor(CsX,CsY,CursorDots);   { Cursor setzen }
     end
      else SetCursor(CsX,CsY,CursorSave); { Cursor aus }
  end;
```

GCursor wird durch die Eingabeschleife des Editors zyklisch aufgerufen und prüft jeweils, ob der »Clock Tick« des Systems bereits den Wert von *NextTick* erreicht hat. Wenn ja, dann ist eine erneute Umschaltung fällig, und *NextTick* bekommt einen Wert, der TICKTIME »Ticks« über der momentanen Systemzeit liegt. Mit dem Wert 6 für TICKTIME ergeben sich ziemlich genau drei Umschaltungen pro Sekunde und damit ein recht angenehmes Erscheinungsbild*).

```
procedure ChangeCursor;  { Dreht den Zustand des Cursors um }
begin  NextTick := SysTicks; GCursor;  end;
```

ChangeCursor setzt einfach den Zeitpunkt der nächsten Umschaltung auf »jetzt« und erzwingt damit ein sofortiges Umschalten des Cursors. Diese Routine wird hauptsächlich vor Operationen aufgerufen, bei denen der Hintergrund wiederhergestellt werden muß.

```
procedure CursorMove(Ch: Char);
begin
  if CursorOn then ChangeCursor;   { Cursor aus }
  case Ch of
   #71: begin  if CsX > EdxMin then Dec(CsX);  { HOME }
               if CsY < EdyMax then Inc(CsY);
        end;
   #72: if CsY < EdYMax then Inc(CsY);         { Aufwärtspfeil }
   #73: begin  if CsX < EdxMax then Inc(CsX);  { PgUp }
               if CsY < EdyMax then Inc(CsY);
         end;
   #75: if CsX > EdxMin then Dec(CsX);         { Linkspfeil }
   #77: if CsX < EdxMax then Inc(CsX);         { Rechtspfeil }
   #79: begin  if CsX > EdxMin then Dec(CsX);  { END }
               if CsY > EdyMin then Dec(CsY);
        end;
   #80: if CsY > EdyMin then Dec(CsY);        { Abwärtspfeil }
   #81: begin  if CsX < EdxMax then Inc(CsX);      { PgDn }
               if CsY > EdyMin then Dec(CsY);
        end;
    end; { case }
   ChangeCursor;  { Cursor wieder auf den Bildschirm }
end;
```

*) Wo habe ich bloß den neunzehnten Band der bundesdeutschen Arbeitsstätten-Richtlinienverordnung hingetan? Zu diesem Thema gibt es garantiert ein ganzes Kapitel.

CursorMove ist mehr oder weniger eine Verlängerung der **case**-Anweisung in der Hauptschleife des Editors und wird auch nur von dort aus aufgerufen, wenn die Prüfung über *ReadKey* ergibt, daß der Benutzer eine Funktionstaste gedrückt hat. Das als *Ch* übergebene Zeichen entspricht dem Scancode dieser Funktionstaste.

```
procedure AddLine(x1,y1,x2,y2:Integer);
var  x:Integer; AddDone: Boolean;
begin
  AddDone := False; x:= 0;
  repeat                { Prüfung: gibt es die Linie schon? }
    Inc(x); with Lines[x] do
    begin
      if ((Xs = x1) and (Ys = y1) and (Xe = x2) and (Ye = y2))
        or ((Xs = x2) and (Ys = y2) and (Xe = x1) and (Ye = y1)) then
        if Kind = LDRAW then AddDone := True  { Linie bereits definiert }
          else if Kind = LDEL then begin  { Linie definiert, aber gelöscht: }
                   Kind := LDRAW; AddDone := True;  { wird wiederhergestellt }
                 end;
    end; { with Lines[x] }
  until (Lines[x].Kind = LDONE) or AddDone;
```

AddLine fügt dem Linien-Array des Editors eine weitere Linie hinzu, deren Start- und Endpunkt als *x1..y2* übergeben werden. *x1..y2* sind relativ zu den Fenstergrenzen des Editors und nicht über MAG skaliert, d.h. entsprechen in ihren Werten den ursprünglichen Vektordefinitionen. Die erste Schleife prüft, ob bereits eine derartige Linie existiert (wenn ja, ignoriert CE den Zeichenbefehl) oder ob sie ursprünglich einmal existierte und zwischenzeitlich gelöscht wurde. Da die Start- und Endpunkte einer einfachen Linie beliebig austauschbar sind (d.h. ein und dieselbe Linie sowohl von links nach rechts als auch von rechts nach links gezeichnet werden kann), fallen die notwendigen Vergleiche etwas umfangreich aus.

```
  x:= 1;
  while not AddDone do
  begin
     with Lines[x] do
     begin
       if Kind = LDONE then   { Ende der Definition erreicht }
       begin
         Lines[x+1] := Lines[x];  Kind := LDEL; { LDONE um eins nach hinten }
        end;
       if Kind = LDEL then begin   { gelöschte Linie }
                            Kind := LDRAW; AddDone := True;
                            Xs:= x1; Ys := y1; Xe := x2; Ye := y2;
                          end;
     end;  { with Lines }
     Inc(x);
  end;  { while }
end; { AddLine}
```

Die zweite Schleife in *AddLine*: wenn eine neue Linie eingetragen werden muß, prüft das Programm zuerst, ob sich vor dem momentanen Ende von *Lines* eine gelöschte Definition befindet. Wenn ja, wird diese Definition mit den neuen Werten überschrieben; wenn nein, wird der LDONE-Eintrag (d.h. die Endmarkierung) um eins nach hinten verschoben und die neue Linie eingefügt. In welcher Reihenfolge die einzelnen Linien im Array *Lines* erscheinen, spielt keine Rolle, weil sie a) sowieso separat gezeichnet werden und b) die Umrechnung in Vektoren den Array-Index nicht als Anordnungskriterium benutzt.

```
function RemoveLine(x1,y1,x2,y2: Integer): Boolean;
var x: Integer;
begin
  x:= 1;
  while (Lines[x].Kind <> LDONE) do
  begin
    with Lines[x] do
      if ((Xs = x1) and (Ys = y1) and (Xe = x2) and (Ye = y2))
        or ((Xs = x2) and (Ys = y2) and (Xe = x1) and (Ye = y1)) then
        if Kind = LDRAW then
        begin  { uff! }
          Kind := LDEL;     { Linie gelöscht }
          RemoveLine := True;
          Exit;
        end;
    Inc(x);
  end; { while }
  RemoveLine := False;  { Linie nicht gefunden! }
end;
```

RemoveLine versucht, die Liniendefinition mit den Start- und Endpunkten *x1..y2* zu löschen und ist im Gegensatz zu *AddLine* eine Funktion: wenn sie das Ergebnis *False* zurückliefert (= »zu löschende Linie nicht gefunden«), dann weiß der Editor, daß er den Löschbefehl ignorieren kann (und keine weiteren Aktionen wie das Neuzeichnen des Editor-Zeichens usw. ausführen muß).

```
procedure SetEndPoint(xp,yp: Integer);
var x: Integer;
begin
  x:= 1;
  while Lines[x].Kind <> LDONE do Inc(x);
  with Lines[x] do
  begin
    PutPixel(WndX+Xs*MAG,WndY-Ys*MAG,0);  { alten Punkt löschen }
    Xs := xp; Ys := Yp;                   { Eintrag neuer Punkt }
   end;
  PutPixel(WndX+Xp*MAG,WndY-Yp*MAG,White); { Zeichnen des neuen Punktes }
end;
```

Die Ausführung des Editor-Befehls *E* (Endpunkt = Zeichenbreite setzen). Dieser Punkt ist als »Startpunkt« des LDONE-Vektors gespeichert, wird an seiner alten Position vom Bildschirm gelöscht und neu gezeichnet. Die Veränderung von *EdCharWidth* (für den zweiten Teil des Vergleichstextes) findet in der Prozedur *DrawCPStr* statt (siehe unten), weil wir zuerst das alte Abbild der zweiten Hälfte dieses Strings vom Bildschirm löschen müssen.

```
procedure DrawCPStr(Func: Integer);
{ Gibt den Vergleichstext in der oberen linken Ecke aus }
var x: Integer;
begin
  SetLineStyle(SolidLn,0,NormWidth); SetColor(White); { normale Linien }
  case func of
   0: begin   { Zeichen zum ersten Mal }
        x := PaintChar(EdCharCX,CYStart,1,False);
      end;
   1: begin { neuer Endpunkt }
        SetViewPort(EdCharCX,0,Edx1-6,Liney1-2,True);
        ClearViewPort;                    { Löschen ab EdCharCX }
        SetViewPort(0,0,GetMaxX,GetMaxY,True);
        EdCharWidth:= PaintChar(EdCharCX,CYStart,1,False);
        CP2Start := EdCharCX+EdCharWidth;
        PutImage(CP2Start,CYStart-CUp-3,CPStrPtr^,NormalPut);
      end;
   2: begin   { Linie löschen }
          SetColor(Black);
          Line(EdCharCX+Spx,CYStart-Spy,
               EdCharCX+CsX,CYStart-CsY);
          SetColor(White);
        end;
   3: begin { Linie zeichnen }
         Line(EdCharCX+Spx,CYStart-Spy,
              EdCharCX+CsX,CYStart-CsY);
       end;
  end; { case }
  SetLineStyle(SolidLn,0,ThickWidth);
end;
```

Die vier möglichen Operationen, die ein »Update« des Vergleichstextes und des darin enthaltenen Editor-Zeichens notwendig machen: Erste Ausgabe beim Start des Editors, Veränderung des Endpunktes und Löschen oder Zeichnen einer neuen Linie. Aufgerufen wird *DrawCPStr* nur durch die Prozedur *Editor* - dort allerdings von mehreren Punkten aus.

```
procedure Editor;
begin
  EdDone := False; SpSet := False;
  CursorOn := False; NextTick := SysTicks;
  if CPStr2Width = 0 then ClearCPWindow;
  DrawCPStr(1); DrawCPStr(0);
  ClearHelpWindow;
  OutTextFormat(Liney2+8, 'Pfeiltasten, HOME, END usw. zur Bewegung des '
                + 'Cursors, ESC = Ende'+ #13
                + '<Space> markiert Startpunkt (SP)'+ #13
                + 'L löscht von SP bis Cursorpos.'+ #13
                + 'Z zeichnet von SP bis Cursorpos.' + #13
                + 'E setzt Endpunkt (= Breite).' + #13);
```

Der Start des Editors: Weder der blinkende Cursor (*CursonOn*) noch die Markierung eines Startpunkts (*SpSet*) sind sichtbar. Der Wert für die nächste Umschaltung des Cursors (Variable *NextTick*) wird auf die momentane Systemzeit gesetzt - der Zustand des Cursors wechselt damit beim ersten Durchlauf der Hauptschleife von *Off* auf *On*.

Die Aufrufe von *DrawCPStr* bringen den Vergleichstext (zusammen mit einem eventuell gewechselten Editor-Zeichen) auf den Bildschirm, über *OutTextFormat* wird die Hilfestellung des Editors ausgegeben.

Wie zu befürchten, besteht der Rest von *Editor* aus einer großen **case**-Anweisung:

```
repeat
  if not KeyPressed then GCursor   { Blink Blink Blink ... }
    else case UpCase(ReadKey) of
         ' ' : begin              { Startpunkt setzen/löschen }
                 if SpSet then    { Startpunkt löschen }
                 begin
                   if CursorOn then ChangeCursor;
                   SetCursor(Spx,Spy,SpSave);
                 end
                  else begin
                        Spx := CsX; Spy := CsY;  { Startpunkt setzen }
                        if not CursorOn then ChangeCursor;
                        SpSave := CursorSave; CursorSave := CursorDots;
                      end;
                  SpSet := not SpSet;
                  end;

         'E' : begin                 { Endpunkt setzen }
                 if CursorOn then ChangeCursor;
                 SetEndPoint(CsX,CsY);
                 ChangeCursor;
                 DrawCPStr(1);
               end;
```

```
          'L' : if SpSet then        { Löschen }
                  if RemoveLine(SpX,SpY,CsX,CsY) then  { Linie vorhanden? }
                  begin
                    if CursorOn then ChangeCursor; { Cursor aus }
                    SetCursor(SpX,SpY,SpSave);     { Start-Cursor aus }
                    SetColor(Black);
                    Line(WndX+Spx*MAG,WndY-Spy*MAG,  { Linie löschen }
                         WndX+CsX*MAG,WndY-CsY*MAG);
                    SpSet := False;    { Startpunkt zurück }
                    SetColor(White);
    { (siehe Text) } EdCharWidth:= PaintChar(WndX,WndY,MAG,True);
                    DrawCPStr(2);  { Löschen im Vergleichstext }
                  end;
```

Der Aufruf von *PaintChar* zeichnet das Editor-Zeichen komplett neu (bis auf die zuvor gelöschte Linie). Diese Zeichenaktion bleibt größtenteils unsichtbar und beseitigt eventuelle Löcher, die durch die Löschung der Linie auf Schnittpunkten mit anderen Linien erzeugt wurden.

Die Löschung im Vergleichstext geschieht dagegen ohne ein anschließendes Neuzeichnen - wen das stört, der kann einen Aufruf von *PaintChar* (mit *EdCharX,CYStart,1,False*) in den entsprechenden case-Zweig von *DrawCPStr* hineinsetzen.

```
          'Z' : if SpSet then        { Zeichnen }
                begin
                  AddLine(Spx,Spy,CsX,CsY);      { neue Linie dazu }
                  if CursorOn then ChangeCursor; { Cursor aus }
                  SetCursor(Spx,Spy,SpSave);     { Start-Cursor aus }
                  Line(WndX+Spx*MAG,WndY-Spy*MAG,WndX+CsX*MAG,WndY-CsY*MAG);
                  SpSet := False;  { Startpunkt zurücksetzen }
                  DrawCPStr(3);    { Zeichnen im Vergleichstext }
                end;
```

Die Definition einer neuen Linie: hier ist außer dem Zeichnen selbst und dem Eintrag in *Lines* (via *AddLine*) kein weiteres »Update« nötig - die Linie wird sogar dann gezeichnet, wenn sie bereits existiert (und *AddLine* überhaupt keinen Eintrag vornimmt).

```
          #27 : EdDone := True;

          #0 : CursorMove(ReadKey);  { Funktionstaste }
        end; { case }
  until EdDone;
  if CursorOn then ChangeCursor; { Ende: Cursor aus! }
end;

{ ******************************************************** }
     { kein Initialisierungsteil }
end.
```

Das Hauptprogramm: CE.PAS

```
program CharEdit;   { CE.PAS, 12/12/87, 14/12/88 as }
uses Crt, Graph, CEGlobal, CEUtils, CEEdit;
var
    x: Integer; Ch: Char; NumStr: string[4];   { Temps }
    EdChange, CEDone, ZDone, SDone: Boolean;   { für die Menüs }
    CharCode: Integer;
```

Das Hauptprogramm definiert außer einigen *Booleans* für **repeat..until**-Schleifen der Untermenüs und einigen temporären Variablen keine weiteren Daten.

```
function EnterChar(px,py: Integer): Char;
var code: Integer; x: LongInt; s:string;
begin
  ClearHelpWindow;
  OutTextFormat(Liney2+8, 'Ein Zeichen im Bereich von '' '' .. ''~'' oder '
                + 'zuerst ein Doppelkreuz (#) und danach den ASCII-Code '
                + '(maximal #127).'+#13+ 'Abschluß der Eingabe mit RETURN, '
                + 'Abbruch mit ESC.');
  MoveTo(px,py);
  OutText('Welches Zeichen? '); GetLine(s,4);  { maximal 4 Zeichen }
  EnterChar := #0;   { Annahme: "ungültig" }
  if (Length(s) = 1) and (s[1] <= '~') then EnterChar := s[1]
    else if (Length(s) > 1) and (s[1] = '#') then
    begin
      Val(Copy(s,2,255),x,code);
      if (code = 0) and (x > 0) and (x < 127) then
        EnterChar := chr(x);
    end;
end;
```

EnterChar wird von allen Menüs verwendet, die ein einzelnes Zeichen vom Benutzer anfordern. Die Eingabe kann entweder direkt oder in der Form *#nnn* erfolgen (wobei *nnn* als ASCII-Code interpretiert wird). Beim Abschluß mit ESC anstelle von RETURN liefert die Funktion das Zeichen NUL zurück.

```
procedure NewChar;
begin
  OutTextXY(5,Liney1+48,'Neue Zeichendefinition');
  with Lines[1] do
  begin
    Kind := LDONE;          { die erste Linie auf "ist die letzte Linie" }
    Xs:= 10; Ys:= 0;        { angenommene Zeichenbreite: 10 }
  end;
  ClearEditWindow;
  EdCharWidth:= PaintChar(WndX,WndY,MAG,True); { zeichnet den Endpunkt }
  EdChar := #$FF; CsX:= 0; CsY:= 0;        { Editor-Cursor }
  Editor;
end;
```

Die Prozedur *NewChar* löscht den Arbeitspuffer des Editors, indem sie der ersten Liniendefinition den Funktionscode LDONE gibt und eine (willkürliche) Zeichenbreite von 10 setzt.

```
procedure ShowCharSet;
{ Gibt den gesamten Zeichensatz im Edit-Fenster aus }
var x,                    { Zähler }
    LineStep: Integer;    { Zeilenhöhe in Pixeln }
    Px,Py: Integer;       { Startpunkt für die jeweils nächste Ausgabe
                            eines Zeichens }
begin
  ClearEditWindow;
  SetLineStyle(SolidLn, 0, NormWidth);  { "dünne" Linien }
  LineStep := CUp-CDown+5;       { Zeilenabstand }
  Px := Edx1 + MAG;              { linker Rand }
  Py := Edy1 + CUp + 5 + MAG;    { Oberkante }

  for x:= MinChar to MinChar + TblSize - 1 do
  begin
    PaintfromVecs(FileBuf[GetCharIndex(Chr(x))],Px,Py);   { Zeichnen }
    Px := GetX;
    if Px > Edx2 - 20 then    { rechter Rand erreicht: neue Zeile }
    begin
      Px := Edx1+MAG;       { zurück zum linken Rand }
      Inc(Py,LineStep);     { nächste Zeile }
      if Py-CDown > Edy2 - MAG - 10 then
      begin
        OutTextError('');      { unterer Rand erreicht: 'Taste drücken... ' }
        ClearEditWindow;
        Py := Edy1 + CUP + 5 + MAG;
      end;
    end;
  end;
  OutTextError('');  { 'Weiter mit einem beliebigen Tastendruck' }
  SetLineStyle(SolidLn, 0, ThickWidth);
  ClearEditWindow;
end;
```

Die Ausführung des Menüpunkts »Tabelle«, d.h. die Ausgabe des gesamten Zeichensatzes in der Reihenfolge seiner ASCII-Codes im Fenster des Editors*). *OutTextError* wird hier verschiedentlich mit einem Nullstring aufgerufen und gibt deshalb nur die Meldung »Weiter mit einem Tastendruck« aus.

*) »Tabelle« ist übrigens eine gute Prüfmöglichkeit für die Funktionsfähigkeit von CE: Wenn die Zeichendefinitionen hier wohlgeordnet erscheinen, kann man guten Gewissens davon ausgehen, daß sämtliche Eckwerte der .CHR-Datei richtig gelesen und interpretiert worden sind.

Normalerweise passen die Zeichensätze auf einmal in das Editor-Fenster hinein - ein Umblättern ist nur bei der Kombination »CGA-Karte und GOTH.CHR« notwendig.

```
function SaveCHRFile: Boolean;
var Ch: Char;
begin
  SaveCHRFile := False;
  HelpESCToMenu; { 'ESC = -> Hauptmenü' }
  MoveTo(5,Liney1+38); OutText('Dateiname: ');
  MoveTo(5,Liney1+48); GetLine(ChrFName,34);
  if ChrFName = '' then Exit;  { Abbruch mit FALSE }
  if Pos('.',ChrFName) = 0 then ChrFName := ChrFName + '.CHR';
  Assign(CHRFile,ChrFName);
  {$I-} Reset(CHRFile); {$I+}   { Prüfung: existiert die Datei bereits? }
  if IOResult = 0 then
  begin
    ClearHelpWindow;
    OutTextFormat(Liney2+8,'Datei '+ ChrFName + ' existiert bereits. ');
    MoveTo(5,Liney2+48); OutText('Überschreiben? (J/N): ');
    repeat
      Ch:= UpCase(ReadKey);
    until (Ch in ['J','N',#27]);
    if Ch = #27 then Ch := 'N';
    OutText(Ch);
    if Ch <> 'J' then
    begin
      Delay(500); Exit;
    end;
    Close(CHRFile);
  end;
  {$I-} Rewrite(CHRFile,1); {$I+}
  Str(IOResult,NumStr);    { auch eine Möglichkeit, IOResult aufzurufen...}
  if NumStr <> '0' then
  begin
    OutTextError('DOS-Fehler '+NumStr+'. ');
    Exit;
  end;
  BlockWrite(CHRFile,FileBuf,FLength);  { keine Prüfung auf "Disk voll" }
  Close(CHRFile);
  SaveCHRFile := True;
end;
```

SaveCHRFile fragt nach einem Dateinamen und schreibt den Inhalt von *FileBuf* unter diesem Namen auf die Diskette, wobei gegebenenfalls zurückgefragt wird, ob eine existierende Datei überschrieben werden soll.

```
function AskCEEnd: Boolean;
begin
  if not EdChange then
  begin
    AskCEEnd := True;  Exit; { es gibt nichts zu speichern }
  end;
  ClearMenuWindow;
  HelpESCtoMenu;   { 'ESC = Zurück zum Hauptmenü' }
  OutTextXY(5,Lineyl+8, 'Ende:');
  OutTextXY(5,Lineyl+18, 'Zeichensatz ist verändert worden.');
  MoveTo(5,Lineyl+28); OutText ('Speichern? (J/N) ');
  repeat
    Ch:= UpCase(ReadKey);
  until (Ch in ['J','N',#27]);
  if Ch <> 'J' then
  begin
    AskCEEnd := (Ch='N'); Exit;  { Abbruch bzw. 'N', zurück zum Hauptmenü }
  end;
  OutText(Ch);  { Ausgabe der Antwort (immer 'J') }
  AskCEEnd := SaveCHRFile;  { Anfrage nach dem Dateinamen und Speicherung }
end;
```

AskCEEnd wird bei der Auswahl von »Ende« aufgerufen und liefert *True* zurück, wenn keine Veränderungen in *FileBuf* vorgenommen wurden. Ansonsten wird das Ergebnis dieser Funktion von *SaveCHRFile* bestimmt (d.h. ist davon abhängig, ob eine Speicherung fehlerfrei ausgeführt werden konnte oder der Benutzer sich für »Nein« entschieden hat).

```
function CheckCHRFile(var BufPtr; var DataStart, FLPos: Integer;
                      FileLen: LongInt): Boolean;
type IBufType = array[0..511] of Byte;
var  x: Integer; IBuf: IBufType;
begin
  IBuf := IBufType(BufPtr);  { spart diverse explizite Typ-Umwandlungen... }
  x := 0;    { Suche nach dem ^Z des Copyrights }
  while (IBuf[x] <> $1A) and (x < $80) do Inc(x);
  if x = $80 then
  begin
    CheckCHRFile := False; Exit;
  end;
  Inc(x);
  DataStart := IBuf[x] + IBuf[x+1] shl 8; { Start "eigentliche" Daten }
  Inc(x,6);
  FLPos := x;  { Position der Anzahl zu verschiebender Bytes }
  if IBuf[x] + IBuf[x+1] shl 8 + DATAOffs <> FileLen then
   CheckCHRFile := False
  else CheckCHRFile := True;
end;
```

Die Prüfung des Dateiformats und das Setzen verschiedener Eckwerte wurde hier in eine eigene Funktion verbannt*): *CheckCHRFile* liefert nur dann das Ergebnis *True*, wenn der Start der eigentlichen Daten (hier: *DataStart*) plus der »Verschiebungslänge« mit der Gesamtlänge der Datei identisch ist. Sozusagen als Nebeneffekt werden bei dieser Prüfung die (vom Hauptprogramm übergebenen) globalen Variablen *DATAOffs* und *FLengthPos* gesetzt.

```
{ ******************************************************************* }
{ ******************************************************************* }

begin
  Writeln('       *** CHAREDIT *** (c) as 12/87, 12/88');
            { ^^ Bitte drinlassen, ja? ^^ }
  if ParamCount <> 1 then  { ... kein .CHR-Dateiname angegeben }
  begin
     Writeln('Bearbeiten von Grafik-Zeichensätzen für Turbo Pascal 4.0/5.0');
     Writeln('Aufruf: CharEdit <Dateiname>[.CHR]');
     Halt(1);
   end;

  { Falls kein Suffix angegeben: Anhängen von '.CHR' }
  ChrFName := ParamStr(1);
  if Pos('.',ChrFName) = 0 then ChrFName := ChrFName + '.CHR';

  { Einlesen der .CHR-Datei }
  Assign(CHRFile, ChrFName); {$I-} Reset(CHRFile,1); {$I+}
  if IOResult <> 0 then
  begin
    Writeln('Nicht gefunden: ', ChrFName);
    Halt(1);
   end;
  Writeln(ChrFName, ' wird eingelesen... ');
  FLength := FileSize(CHRFile); BlockRead(CHRFile,FileBuf,FLength);
  Close(CHRFile);

  { Format-Prüfung - setzt DATAOffs und FLengthPos }
    if not CheckCHRFile(FileBuf[0], DATAOffs, FLengthPos, FLength) then
    begin
      Writeln('Datei ', CHRFName,  ' ist nicht im erwarteten Format!');
      Halt(1);
    end;
```

*) ... was nur im Zusammenhang mit der Routine *ImportChar* einen Sinn macht (siehe Seite 396). Dieser Routine haben wir auch den untypisierten Parameter *Buf* zu verdanken, weil sie mit einem eigenen Puffer für die zweite .CHR-Datei arbeiten muß.

```
{ Ermittlung der Eckdaten des Zeichensatzes }
TBStart := DATAOffs + $10;          { Start der Vektor-Adresstabelle }
TblSize:= FileBuf[DATAOffs+1];      { Anzahl der Definitionen + 1 }
WidthTbl := TBStart + TblSize shl 1; { Start der Breitentabelle }
MinChar := FileBuf[DATAOffs+4];     { ASCII-Code der ersten Definition }
VecStart := FileBuf[DATAOffs+5]
            + FileBuf[DATAOffs+6] shl 8;    { Vektoren-Start }
CUp := FileBuf[DATAOffs+8];           { Zeichenhöhe für Großbuchstaben }
CDown := ShortInt(FileBuf[DATAOffs+$0A]);   { Unterlänge }
```

Zu diesem Teil des Hauptprogramms gibt es eigentlich nicht mehr viel zu sagen - bis auf den Aufruf von *CheckCHRFile* anstelle einer direkten Prüfung ist er mit dem Programm *DrawChar* so gut wie identisch.

```
{ Start der Grafik }
GraphDriver:= Detect; InitGraph(GraphDriver, GraphMode,'');
if GraphResult <> 0 then
begin
  CloseGraph;
  Writeln(GraphErrorMsg(GraphDriver));
  Halt(1);
end;
if GraphDriver = CGA then SetGraphMode(CGAHi);  { wir brauchen 640 Pixel! }

{ Ermittlung der Eckwerte für das Layout des Bildschirms }
MAG := (GetMaxY + 1) div (CUp + 6 - CDown);  { Vergrößerungsfaktor }
WndX := 300; WndY := (CUp+5) * MAG;

{ Das grafische Leerzeichen für GetLine }
GetMem(Blank,ImageSize(0,0,8,14)); GetImage(0,0,8,14,Blank^);

{ Der Rahmen des Editors und die Grenzen der Cursorbewegung }
Edx1 := WndX-MAG; Edx2:= GetMaxX;
Edy1 := 0; Edy2 := WndY - (CDown-1) * MAG;
EdxMin := 0; EdxMax := (GetMaxX-WndX) div MAG;
EdyMax := CUp + 4; EdyMin := CDown;

{ Der Rahmen der Menüs }
CXStart := 5; CYStart:= CUp + 8;  { "Vergleichstext" }
Liney1 := CYStart - CDown + 8;
Liney2 := (GetMaxY - Liney1) div 2 + Liney1;

SetColor(Blue); SetLineStyle(SolidLn,0,ThickWidth);
Line(Edx1,Edy1,Edx2,Edy1);   { Oberkante }
Line(Edx1,Edy1,Edx1,Edy2);   { linker Rand }
Line(Edx1,Edy2,Edx2,Edy2);   { Unterkante }
Line(0,Liney1,Edx1,Liney1);
Line(0,Liney2,Edx1,Liney2);
```

```
{ Die Markierungen innerhalb der Fenster }
SetColor(Green);
Line(Edx1-5, WndY-CUp*MAG, Edx1+5, WndY-CUp*MAG);
Line(Edx1-5 , WndY, Edx1+5, WndY);
Line(Edx1-5, WndY-CDown*MAG, Edx1+5, WndY-CDown*MAG);
SetColor(White);

{ Der Vergleichstext }
CPStr := ''; CP2Start := 0; CPStr2Width := 0;
CPStrPtr := Blank; EdCharCX := 0;

EdChar := #0; EdCharWidth := 0;      { kein Zeichen im Editor }
CEDone := False; EdChange := False; { Datei unverändert }
CsX:= 0; CsY:= 0;                   { Cursorposition im Editor }
```

Der Start der Grafik und die Skalierung abhängig von den Grenzen des Bildschirms bzw. dem Platzbedarf des Zeichensatzes. Die Rahmen der einzelnen Fenster sind statisch, d.h. werden nur einmal während des gesamten Programmlaufs gezeichnet.

Die Hauptschleife von CE besteht zu 99,9% aus einer wahrhaft monströsen case-Anweisung, in die die einzelnen Untermenüs eingeschachtelt sind*):

```
{ ****************** Hauptschleife des Programms ************** }
repeat
  ClearMenuWindow;
  ClearHelpWindow;
  OutTextXY(5,Lineyl+8,'Speichern  Tabelle  Zeichen  Ende');
  case UpCase(ReadKey) of
    'S': begin { Speichern der Datei }
           ClearMenuWindow;
           OutTextXY(5,Lineyl+8,'Speichern');
           if SaveCHRFile then EdChange := False; { Speichern der Datei }
         end;
    'T': begin   { Tabelle aller definierten Zeichen }
           ShowCharSet;
           if EdChar <> #0 then          { neu zeichnen }
             EdCharWidth := PaintChar(WndX,WndY,MAG,True);
         end;
    'E': CEDone := AskCEEnd; { Ende von CharEdit }
```

Die drei einfachen Auswahlmöglichkeiten des Hauptmenüs: *Speichern*, *Tabelle* und *Ende*.

*) Hier wird man sich wahrscheinlich mit **begin** und **end** etwas schwerer tun - das ist meiner Meinung nach aber immer noch kein Grund für die Aufteilung in ein halbes Dutzend Prozeduren, die alle nur einmal aufgerufen werden, und die man erst einmal im Listing zusammensuchen muß, um etwas über den Ablauf des Programms zu erfahren. Anders gesagt: Wenn ich **case**-Anweisungen dieser Größe für »schlechten Programmierstil« halten würde, hätte ich das Programm anders aufgebaut.

```
'Z': begin   { Untermenü 'Zeichen' }
       ZDone := False;
       repeat
         ClearMenuWindow;
         HelpESCtoMenu;  { 'ESC = Zurück zum Hauptmenü' }
         OutTextFormat(Lineyl+8,'Bearbeiten   Speichern als' + #13
                              + 'Fortsetzen   Löschen'+ #13
                              + 'Vergleich    Neu');

         case UpCase(ReadKey) of
           'B': begin  { Zeichen bearbeiten }
                  Ch := EnterChar(5, Lineyl+38);
                  if (Ch <> #0 ) and (Ch < Chr(MinChar))
                   then NewChar
                   else if Ch <> #0 then
                   begin
                     EdChar := Ch;
                     VecstoLines(Filebuf[GetCharIndex(EdChar)]);
                     ClearEditWindow;
                     EdCharWidth := PaintChar(WndX,WndY,MAG,True);
                     CsX:= 0; CsY:= 0;  { Editor-Cursor }
                     Editor;
                   end;
                end;
```

Der Wahlpunkt *Zeichen* des Hauptmenüs und die Funktion *Bearbeiten*: *Zeichen* verwendet eine eigene Schleife (mit einer eigenen boolschen Variablen) und bringt als erstes das entsprechende Menü auf den Bildschirm.

Bearbeiten fordert vom Benutzer die Eingabe eines Zeichens an und setzt dieses Zeichen nur dann als »Editor-Zeichen«, wenn die Eingabe gültig ist bzw. nicht abgebrochen wurde - ansonsten bleibt der Arbeitspuffer des Editors unverändert. Bei einer gültigen Eingabe wird der Arbeitspuffer des Editors über *VecsToLines* mit den Äquivalenten der entsprechenden Vektordefinition gefüllt und schließlich der Editor aufgerufen.

```
           'N': NewChar; { neues Zeichen }

           'L': begin  { Zeichendefinition löschen }
                  ClearMenuWindow;
                  OutTextXY(5,Lineyl+8, 'Zeichendefinition löschen');
                  CharCode := Ord(EnterChar(5,Lineyl+18));
                  if (CharCode <> 0) then { nicht mit ESC abgebrochen }
                   begin
                    if CharCode < MinChar
                     then OutTextError
                          ('Dieses Zeichen ist nicht definiert. ')
```

```
        else begin
         DelDefinition(CharCode);
         Str(CharCode,NumStr);
         OutTextFormat(Liney1+38,
         'Zeichendefinition '+ NumStr + ' gelöscht.');
         OutTextError(''); EdChange := True;
       end; { else CharCode < MinChar }
     end; { if CharCode <> 0 }
    end; { case 'L' }
```

Die Funktionen *Neu* und *Löschen*: *Neu* besteht aus dem Aufruf der Routine *NewChar*, die den Arbeitspuffer des Editors löscht und nach der Ausgabe der Meldung »Neues Zeichen« den Editor aufruft.

Löschen fordert die Eingabe des zu löschenden Zeichens über *EnterChar* an (und führt keine weiteren Operationen aus, wenn diese Eingabe mit ESC abgebrochen wurde). Ansonsten geht es mit einem Aufruf von *DelDefinition* (im Unit *CEUtils*) weiter, dem die Ausgabe des ASCII-Codes des gelöschten Zeichens folgt. *EdChange* wird hier auf *True* gesetzt - die .CHR-Datei wurde durch die Löschung verändert.

```
'S': if EdChar <> #0 then
     begin  { Speichern als }
       SDone := False;
       repeat
       ClearMenuWindow;  OutTextFormat(Liney1+8,
             'Zeichen überschreiben' + #13 + 'Neu anfügen');
       ClearHelpWindow;  OutTextXY(5,Liney2+8,
             'ESC = Zurück zum Menü ''Zeichen''');
```

Die Funktion *Speichern als* definiert ein eigenes Untermenü mit den Möglichkeiten *Zeichen überschreiben* und *Neu anfügen*:

```
case UpCase(ReadKey) of
  #27: SDone := True;
  'Z': begin  { Existierendes Zeichen }
       ClearMenuWindow;
       OutTextXY(5,Liney1+8,'Zeichen überschreiben');
         CharCode := Ord(EnterChar(5,Liney1+28));
         if CharCode = 0 then SDone := True
           else if CharCode < MinChar then
           OutTextError
             ('Dieses Zeichen ist nicht definiert. ')
            else begin
              DelDefinition(CharCode);
              InsertDefinition(CharCode);
              SDone := True; EdChange := True;
            end;
         end;
```

Das Überschreiben eines Zeichens besteht (wie ganz zu Anfang der Besprechung schon einmal angedeutet) aus aufeinanderfolgenden Aufrufen von *DelDefinition* (löscht die alte Zeichendefinition bzw. verkürzt sie auf einen einzigen LDONE-Vektor) und *InsertDefinition* (fügt die neue Definition ein bzw. erweitert den LDONE-Vektor auf die neue Definition). Weil diese beiden Operationen die .CHR-Datei verändern, wird danach das Flag *EdChange* gesetzt.

```
'N': begin  { Neu anfügen }
       ClearMenuWindow;
       OutTextXY(5,Liney1+8,'Neu anfügen');
       SDone:= True;
       CharCode := MinChar -1;
       if CharCode < 0 then
         OutTextError('Zeichentabelle ist voll!')
        else
        begin
          Str(CharCode,NumStr);
          InsertDefinition(CharCode);
          OutTextFormat(Liney1+28,
          'Das Zeichen hat den ASCII-Code '
          + NumStr + ' zugewiesen bekommen.');
          OutTextError('');
          EdChange := True;
        end;
     end;
   end; { case }
 until SDone;
end;
```

Beim Anfügen einer Zeichendefinition muß die Möglichkeit berücksichtigt werden, daß sich die Tabelle nicht mehr weiter in Richtung auf niedrigere ASCII-Codes erweitern läßt*).

```
'F': if EdChar <> #0 then Editor;  { Fortsetzen }
'V': begin  { Vergleich }
       ClearHelpWindow;
       OutTextFormat(Liney2+8,
            'Ein ''^'' steht hier für das zu bearbeitende '
          + 'Zeichen- ''e'' im Editor und ''T^st'' als '
          + 'Vergleichs-String ergibt ''Test''. ''^'' darf '
          + 'nur einmal in diesem String vorkommen.'+ #13
          + 'Abschluß mit RETURN, Abbruch mit ESC.');
       MoveTo(5,Liney1+38);
       OutText('Vergleichstext: '); GetCPStr;
     end;
```

*) Was beim Bearbeiten von SANS.CHR praktisch sofort der Fall ist - dieser Zeichensatz läßt sich nur erweitern, indem man einige der Spielkarten im ASCII-Bereich von 01..31 überschreibt.

Hier befinden wir uns wieder auf der Ebene des Menüs *Zeichen*: Die Auswahl von *Fortsetzen* aktiviert den Editor direkt (wenn bereits irgendeine Definition in den Arbeitspuffer geladen wurde).

Die Eingabe des Vergleichstextes besteht im Hauptmenü mehr oder weniger aus der Ausgabe der Hilfestellung - die Hauptarbeit wird von *GetCPStr* (im Unit *CE-Edit*) erledigt.

```
                #27: ZDone := True;
              end; { case }
            until ZDone;  { Ende des Menüs 'Zeichen' }
          end; { case Z im Hauptmenü }
      end; { case-Anweisung des Hauptmenüs }
    until CEDone;
    CloseGraph;
  end.
```

Die Enden der diversen Schleifen, case-Anweisungen und schließlich das Ende des gesamten Programms.

Lesen anderer Zeichensätze: Die Prozedur *ImportChar*

Eine Mixtur verschiedener Zeichensätze in einer Datei zeitigt in den meisten Fällen recht merkwürdige (wie eine Spielkarte aus SANS in GOTH) bis völlig abstruse Ergebnisse (Buchstaben aus LITT haben z.B. in den anderen Zeichensätzen ungefähr die Höhe eines Kommas). Trotzdem kann eine »Import«-Funktion sinnvoll sein - das beste Beispiel für eine mögliche Anwendung sind meine eigenen bitteren Erfahrungen, die etwa so aussehen:

Man erweitere den Zeichensatz GOTH mit einer guten Stunde Arbeit um die deutschen Umlaute - ohne ihn zwischendurch zu speichern, versteht sich - und lösche dann versehentlich das Dollarzeichen*). Ohne eine »Import«-Funktion bleiben nun drei Möglichkeiten:

- das versehentlich gelöschte/demolierte/überschriebene Zeichen wird »zu Fuß« neu eingegeben;
- Neustart von CE zusammen mit der Originaldatei (sprich: das Ganze noch einmal von vorne);
- Dollarzeichen? Wer braucht sowas?

Mit einer »Import«-Funktion fallen solche Reparaturen recht geradlinig aus: Weil CE die gesamte .CHR-Datei beim Start *en bloc* liest und danach nicht mehr auf die Diskette zugreift, gilt auch der Original-Zeichensatz als »andere Datei« - im zuvor geschilderten Unfall wird also einfach das Zeichen »$« aus der Datei GOTH »importiert«.

*) Wie um Himmels Willen ich das geschafft habe? Hauptsächlich mit CAPS LOCK.

Soviel zur Vorgeschichte und als Begründung für den nachträglichen Einbau dieser Funktion. *ImportChar* findet sich in der erweiterten Version von CE im Untermenü *Zeichen* und besteht aus den folgenden Schritten:

- Eingabe des Namens einer Zeichensatz-Datei (das Suffix .CHR wird nötigenfalls automatisch angehängt);
- Format-Prüfung dieser Datei und Ermittlung der Eckwerte via *CheckCHRFile*;
- Eingabe des zu lesenden Zeichens über *EnterChar* (d.h. entweder als einzelnes Zeichen oder in der Form *#nnn*);
- Einlesen der entsprechenden Zeichendefinition aus dieser Datei in den Puffer des Editors (der dabei ohne Rückfrage überschrieben wird).

Von hier aus kann man das gelesene Zeichen mit *Speichern als* in die .CHR-Datei einfügen, die beim Start von CE geladen wurde.

Wenn Sie weitere Zeichen übertragen wollen, müssen Sie das gesamte procedere wiederholen (was nicht gerade sonderlich komfortabel ist - aber für die Übertragung ganzer Zeichensätze war die Funktion auch nicht gedacht).

Vor dem Sprung in die Details eine Entschuldigung - und zugleich eine

WARNUNG! Weil *ImportChar* definitiv eine nachträgliche Erfindung ist und als solche das Original so wenig wie möglich stören sollte, führt diese Routine den gesamten Analyseprozeß von CE in Eigenregie aus - und ist (ohne Erläuterungen) trotzdem nicht länger als eine Buchseite. Womit a) der Beweis erbracht wäre, daß sich in Turbo Pascal genauso kompakte Programme formulieren lassen wie in Turbo C*) und b) dabei auch genau derselbe Stacheldrahtverhau entsteht.

Zur Sache. Für den Aufruf von *ImportChar* wird das Untermenü *Zeichen* in der Hauptschleife des Programms (Datei CE.PAS) um zwei Zeilen erweitert:

```
.....
'Z': begin   { Untermenü 'Zeichen' }
       ZDone := False;
       repeat
         ClearMenuWindow;
         HelpESCtoMenu;  { 'ESC = Zurück zum Hauptmenü' }
         OutTextFormat(Lineyl+8,'Bearbeiten   Speichern als' + #13
                            + 'Fortsetzen   Löschen'+ #13
                            + 'Vergleich    Neu      Import');
         case UpCase(ReadKey) of
           'I': ImportChar;  { Import aus anderer Datei }
           'B': begin  { Zeichen bearbeiten }
                  Ch := EnterChar(5, Lineyl+38);
                .....
```

*) Aus diesen Gefilden stammt das gute Stück nämlich - ursprünglich war es einmal eine Dreingabe von *Turbo C: Konzepte...* gegenüber *Turbo Pascal 4.0*.

Die Routine selbst sollte direkt vor das Hauptprogramm gesetzt werden. Modifikationen der anderen Module von CE und/oder weiterer Programmteile in CE.PAS sind nicht notwendig.

Das Listing:

```
procedure ImportChar;
type
  CHRDataStruct = record
                      x1: Byte;     { [0] - $2B - unbekannter Zweck }
                 TblSize: Integer; { [1] - Anzahl Definitionen - 1 }
                      x2: Byte;     { [3] - 00 }
                 MinChar: Byte;     { [4] - niedrigster ASCII-Code }
                VecStart: Integer; { [5] - Start der Vektordefinitionen }
                      x3: Char;     { [7] - .. }
                     CUp: ShortInt;{ [8] - Höhe für Großbuchstaben }
                      x4: Char;     { [9] - .. }
                   CDown: ShortInt;{ [10] - Unterlänge }
              end;
   WPtrType = ^Word;    { zum Lesen von Werten mit 16 Bit }

var
 ICHRInfo: ^CHRDataStruct;     { Zeiger auf die Kopfinformation }
 IFile: File of Byte;          { aus dieser .CHR-Datei wird gelesen }
 IBuf: array[0..511] of Byte;    { Puffer zum Lesen der Daten }
 Ch: Byte;
 IDOffs, IDummy: Integer;      { DATAOffs & eine "Dummy"-Variable }
```

Anstatt einen weiteren Satz von Variablen für die Eckwerte einer Zeichensatz-Datei zu deklarieren, begnügt sich *ImportChar* mit einem entsprechend typisierten Zeiger, der später auf den (temporär in *IBuf* eingelesenen) Dateikopf gesetzt wird.

```
function ReadIFile: Integer;  { Liest 512 Bytes in IBuf }
var x: Integer;
begin
  x := 0;
  while (x < 512) and not Eof(Ifile) do
  begin
{$I-} Read(Ifile, IBuf[x]);  {$I+  ** kein Dazwischenfunken durch SYSTEM }
    Inc(x);
  end;
  ReadIFile := x;
end;
```

Die lokale Funktion *ReadIFile* wird zweimal benutzt: beim ersten Aufruf liest sie den Dateikopf in *IBuf*, beim zweiten Aufruf die eigentliche Zeichendefinition.

```
begin  { ImportChar }
  ClearHelpWindow;
  OutTextFormat(Liney2+8, 'Lesen einer Zeichendefinition aus einer anderen '
                          + '.CHR-Datei in den Arbeitspuffer des Editors'
                          + #13 + 'ESC -> Zurück zum Hauptmenü');
  OutTextFormat(Liney1+38, 'Dateiname: ');
  Moveto(90,Liney1+38); GetLine(ChrFName, 23);
  if ChrFName = '' then Exit;   { Abbruch mit ESC }
```

Der Benutzer sollte unter kryptischer Programmierung nicht zu leiden haben: *ImportChar* beginnt mit der Ausgabe einer Hilfestellung und fordert danach die Eingabe des Dateinamens an.

```
  if Pos('.', ChrFName) = 0 then ChrFName := ChrFName + '.CHR';
  Assign(IFile, ChrFName); {$I-} Reset(IFile); {$I+}
  if IOResult <> 0 then
  begin
    OutTextError('Datei nicht gefunden. ');
    Exit;
  end;
  if (ReadIFile < 512) or not CheckCHRFile(IBuf[0], IDOffs, IDummy,
     FileSize(IFile)) then
  begin
    OutTextError('Datei nicht im richtigen Format! ');
    Close(Ifile);
    Exit;
  end;
```

Öffnen der Datei (wobei das Suffix .CHR nötigenfalls angehängt wird), Einlesen der ersten 512 Bytes und Prüfung des Dateiformats via *CheckCHRFile*: die »Verschiebungslänge« interessiert hier nicht und wird als »Dummy« übergeben, die Dateilänge während des Aufrufs ermittelt.

```
  Ch := Ord(EnterChar(5, Liney1 + 48));   { "Welches Zeichen?" }
  if Ch = 0 then
  begin  { ESC }
    Close(Ifile); Exit;
  end;
  ICHRInfo := @IBuf[IDOffs];
  with ICHRInfo^ do
  begin
    if (Ch < MinChar) or (Ch >= MinChar + TblSize) then
    begin
      OutTextError('Dieses Zeichen ist nicht definiert. ');
      Close(Ifile);
      Exit;
    end;
```

Eingabe des zu lesenden Zeichens und Prüfung, ob eine Definition des angegebenen ASCII-Codes in der Datei enthalten ist.

```
Seek(Ifile, VecStart + IDOffs +
    WPtrType(@IBuf[(Ch-MinChar) shl 1 + IDOffs + $10])^);
end;  { with ICHRInfo^ }
IDummy := ReadIFile;   { Einlesen der folgenden 512 Bytes }
Close(IFile); { Uff! }
```

Die Adresse innerhalb der Datei, ab der die gesuchte Definition gespeichert ist, ergibt sich auf dieselbe Weise wie beim Auslesen von *FileBuf* - nur daß hier sämtliche Rechenschritte in einem Ausdruck zusammengefaßt sind, der seinerseits als Argument für *Seek* dient. Ab der durch *Seek* gesetzten Position werden 512 Bytes eingelesen - das sollte für die gesamte Definition des Zeichens ausreichend sein.

Vorsicht: *IBuf* enthält ab diesem Zeitpunkt nicht mehr den Kopf der Datei, sondern die gesuchte Vektorfolge - weitere Adressierungen über *ICHRInfo* würden also Unsinn ergeben.

```
  VecsToLines(IBuf[0]);   { Umwandlung in Linien }
  ClearEditWindow;
  EdCharWidth := PaintChar(WndX, WndY, MAG, True); { Zeichnen... }
  CsX := 0; CsY := 0;  { Editor-Cursor }
  EdChar := #$FF;  { "neues Zeichen" bzw. "Editorpuffer gesetzt" }
end;
```

Der Rest von *ImportChar* ist einigermaßen geradlinig: Über den Aufruf von *VecsToLines* mit der Startadresse von *IBuf* als Parameter werden die gelesenen Vektorfolgen in Linien (Array *Lines*) umgewandelt, *PaintChar* gibt danach den Inhalt von *Lines* im Fenster des Editors aus.

Neue Versionen von *OutText*

Die Erläuterungen und Vorschläge dieses Abschnitts gehen davon aus, daß eine oder mehrere Zeichensatz-Dateien um alle deutschen Sonderzeichen erweitert wurden, also in etwa:

#24	#25	#26	#27	#28	#29	#30	#31
Ä	Ö	Ü	§	ä	ö	ü	ß

Um diese Erweiterungen nun auch aus dem Zeichensatz »herauszuholen«, gibt es mehrere Möglichkeiten, von denen die folgende sicher die am wenigsten empfehlenswerte ist:

```
OutText('Gr'#30#31' Gott!');   { <- heißt: 'Grüß Gott!' }
```

Die einfachste verwendbare Möglichkeit besteht in einer Pascal-Routine, die einen String als Parameter erwartet, seinen Inhalt entsprechend umcodiert und danach *Graph.OutText* bzw. *Graph.OutTextXY* aufruft.

Ein erster Ersatz für *Graph.OutText* wurde bereits vor CE auf Seite 353 besprochen - hier folgt eine zweite Version:

```
procedure OutText(s: string);  { Ersatz für Graph.OutText }
var x,y: Integer;
const
  OrgChars : array[1..8] of Char = ('Ä','Ö','Ü',#21,'ä','ö','ü','ß');
  NewChars : array[1..8] of Char = (#24,#25,#26,#27,#28,#29,#30,#31);
begin
  for y := 1 to 8 do
  begin
    repeat
      x := Pos(OrgChars[y],s);
      if x <> 0 then s[x] := NewChars[y];
    until x = 0;
  end;
  Graph.OutText(s);
end;
```

Diese Version arbeitet ein gutes Stück schneller, weil *Pos* einen direkten Prozessorbefehl (SCASB) zur Absuche des Strings verwendet. Dafür ist sie im Gegensatz zur ersten Version für einen Austausch »im Kreis« (A -> B, B -> C, C-> A) nur bedingt verwendbar, was aber im Zusammenhang mit grafischen Zeichensätzen keine allzugroße Rolle spielen dürfte.

Der Zeitbedarf einer Umcodierung ist gegenüber dem von *OutText* so gering, daß sich eine weitere Optimierung in diesem Punkt eigentlich nicht lohnt. Die folgende Version von *OutText* als Maschinenprogramm ist deshalb mehr der Vollständigkeit halber wiedergegeben - noch dazu ist sie nicht ganz unproblematisch, weil sich der Assembler weigert, »qualifizierte« Deklarationen zu akzeptieren:

```
EXTRN Graph.OutText:far     ; <== ergibt eine Fehlermeldung
```

Leider muß die Assembler-Version von *OutText* deshalb über eine zusätzliche Prozedur zu Beginn des Programms mit der Adresse von *Graph.OutText* initialisiert werden.

Das Listing des Maschinenprogramms:

```
; OUTTEXT.ASM - Umcodierung eines Strings für grafische Zeichensätze
; Version mit einer lokalen Stringvariablen, d.h. ohne getarnten var-Parameter

DATA Segment Word public
     EXTRN OutChars: Byte    ; String = 'ÄÖÜäöüß...'+ #24#26#26...
DATA ends

CODE Segment byte public
     PUBLIC OutText
     PUBLIC OutTextInit
     ASSUME cs:CODE, ds:DATA
```

```
; --------------------------------
; procedure OutText(s:string);
; --------------------------------
Init:    db 0
OutText  proc near
         cmp Byte Ptr Init,0 ; Zeiger zu Graph.OutText gesetzt?
         jz OtEnd            ; ... NEIN! -> Exit
         push bp
         mov bp,sp
         cld
; ---------------------------------
         sub sp,0100h        ; Platz für den lokalen String
         mov bx,ds           ; Datensegment
         push ds
         lds si,[bp+4]       ; DS:SI <- Zeiger auf den Parameter s
         push ss
         pop es              ; ES:DI <- Zeiger auf den lokalen String
         lea di,[bp+0FF00h]
         push es
         push di             ; Stringadresse für GRAPH.OUTTEXT
         push es
         push di             ; Stringadresse für die Bearbeitung
         lodsb
         stosb               ; Längenbyte kopieren
         xor ch,ch
         mov cl,al           ; CX = Länge = Zähler
         repnz movsb         ; Kopie des Strings
         pop si
         pop ds              ; DS:SI auf den lokalen String
         mov es,bx           ; ES:DI auf OutChars
         mov di,Offset OutChars
;-------------------------------------------
         mov bl,[si]         ; Längenbyte des Strings (Zähler)
         mov cl,es:[di]      ; Längenbyte von OutChars
         inc di
         shr cl,1            ; nur die erste Hälfte vergleichen
         mov dx,cx           ; Index zur zweiten Hälfte
Loop:    push di
         inc si
         mov al,[si]         ; 1./nächstes Zeichen von s
         mov cx,dx
         repnz scasb         ; Vergleich CX Bytes AL <-> ES:DI, erhöht DI
         jnz Next
         dec di              ; zurück auf das fragliche Zeichen
         add di,dx           ; DI auf die andere Hälfte von OutChars
         mov al,es:[di]      ; neues Zeichen
         mov [si],al         ; -> setzen
```

```
Next:    pop di
         dec bl          ; Längenzähler für den String
         jnz Loop
         push es
         pop ds          ; DS zurück
         db 9Ah          ; CALL FAR
OtCall:  dw 0,0          ; <- Adresse von Graph.OutText
         mov sp,bp
         pop bp
OtEnd:   ret 4
OutText  endp

; --------------------------------
; procedure OutTextInit(p:Pointer);   Setzt die Adresse von Graph.OutText
; --------------------------------   in OtCall
OutTextInit proc near
         push bp
         mov bp,sp
         les di,[bp+4]
         mov word ptr OtCall,di
         mov ax,es
         mov word ptr OtCall+2,ax
         mov byte ptr Init,0FFh     ; Flag: Zeiger initialisiert
         pop bp
         ret 4
OutTextInit endp

CODE     ends
         end
```

Und ein Beispielprogramm in Pascal dazu:

```
program OutTextTest;
uses Graph;
var GraphDriver, GraphMode: Integer;
const                 { alte Codes   |   neue Codes }
  OutChars : string =  'ÄÖÜ'#21'äöüß'#24#25#26#27#28#29#30#31;

procedure OutText(s:string); external;
procedure OutTextInit(p:Pointer); external;
{$L OUTTEXT}

begin
  OutTextInit(@Graph.OutText); { setzt den Aufruf von GRAPH.OUTTEXT }
  GraphDriver := Detect; InitGraph(GraphDriver, GraphMode,'');
  SetTextStyle(GothicFont,HorizDir,6);
  OutText('äöüßÄÖÜ');
  Readln; CloseGraph;
end.
```

Laden und Speichern von Grafikbildern

Es wird sich wohl nie so ganz klären lassen, wieso *Graph* mit allen erdenklichen Funktionen ausgerüstet ist - und jede Möglichkeit zum direkten Laden oder Speichern von Grafikbildern fehlt*). Aus der Marketingabteilung von Borland dürfte diese Idee jedenfalls nicht stammen: Die als »4.0-Version« verkaufte Ausgabe der Graphix Toolbox, die entsprechende Funktionen bietet, arbeitet unabhängig von *Graph* (und um das Kind beim Namen zu nennen: sie ist zu Graph inkompatibel).

Das Unit EGAFILE löst dieses Problem für EGA- und VGA-Karten, wobei nicht nur das momentan gesetzte Bildformat, sondern auch die aktive Speicherseite berücksichtigt wird. Es beschränkt sich nur der Übersichtlichkeit halber auf diese beiden Karten (deren Register-Ansteuerung immer noch so etwas wie schwarze Magie zu sein scheint) - wenn man die case-Anweisungen für die Prüfung der Grafikmodi in eine weitere case-Anweisung zur Prüfung des geladenen Grafiktreibers »einschachtelt«, erhält man universelle (aber auch entsprechend umfangreiche) Funktionen zur Bearbeitung beliebiger Grafikbilder.

```
unit EGAFile;
{ Laden und Speichern von EGA/VGA-Bildern }

interface

uses Graph;

  procedure SetActivePage(PageNo: Word);  { ersetzt GRAPH.SetActivePage }
  function SaveScreen(name: string): Integer;  { = IOResult }
  function LoadScreen(name: string): Integer;  { = IOResult }

{ ********************************************************** }
implementation

const
  EGASelRegs = $3CE;      { Auswahl eines Registers (Index) }
  EGARegIO = $3CF;        { Das jeweils ausgewählte Register }
  ActivePage: Word = 0;   { Hält die Nummer der aktiven Seite fest }

procedure SetActivePage(PageNo: Word);
begin
  ActivePage := PageNo;
  Graph.SetActivePage(PageNo);
end;
```

*) Wenn Sie bei dieser Behauptung an Möglichkeiten wie `GetImage(p, ImageSize(0,0,GetMaxX, GetMaxY));` `Write(XFile, p^);` denken, liegen Sie nur teilweise richtig: Zum einen brauchen *GetImage* und *PutImage* vergleichsweise eine halbe Ewigkeit, zum anderen sind sie auf 64 KByte begrenzt - um ein EGA-Bild zu speichern, müßte man es also in zwei Abschnitte unterteilen. Dazu kommt eine entsprechende Typisierung der Datei...

Graph.SetActivePage muß durch diese Prozedur ersetzt werden, weil es eine Funktion *GetActivePage* nicht gibt - folglich müssen wir die momentan aktive Speicherseite in einer eigenen Variablen festhalten, damit die Routinen zum Lesen und Schreiben wissen, welche Bereiche des Bildspeichers bearbeitet werden sollen.

Achtung! Jedes Programm, das *EGAFile* verwendet, muß dieses Unit **vor** *Graph* mit **uses** angeben, also:

```
(* uses EGAFile, Graph; *)
```

Ansonsten würde der Compiler die Redefinition von *SetActivePage* als lokal zu *Graph* interpretieren - mit der Folge, daß sich Programm-Aufrufe von *SetActivePage* direkt auf *Graph.SetActivePage* beziehen.

```
procedure SetEReg(Regno, Value: Byte);
{ Setzt das durch Regno angegebene EGA-Register mit dem Wert Value }
begin
  Port[EGASelRegs] := Regno;    { Auswahl des Registers }
  Port[EGARegIO]:= Value;       { und Schreiben des Wertes }
end;
```

Die Prozedur zum Setzen eines EGA-Registers. Sie wird unter anderem zur Anwahl der Bildspeicherebene benutzt, die *SaveScreen* als nächstes lesen und auf die Diskette schreiben soll.

```
function SaveScreen(name: string): Integer;  { = IOResult }
var f: File;  { untypisiert }
    Temp, Page: Integer;

function WriteScreen(Count, PageSize: Word): Integer;
{ Schreibt jeweils eine Speicherebene. Funktionsergebnis entspricht IOResult }
var Result: Word;
    ScrPnt: Pointer;
begin
  ScrPnt := Ptr($A000+PageSize * ActivePage, 0);  { Startadresse }
  BlockWrite(f,ScrPnt^,Count,Result);
  if Result <> Count then
  begin
    SaveScreen := 101;
    WriteScreen:= 101;   { simuliert IOResult: Diskette voll }
    Close(f);
  end
   else WriteScreen := 0;
end;

begin  { SaveScreen }
  Assign(f,name); {$I-} Rewrite(f,1); {$I+}
  Temp := IOResult; SaveScreen := Temp;  { Fehler? }
  if Temp <> 0 then Exit;  { Ende mit Ergebnis = IOResult }
```

```
  case GetGraphMode of
    EGALo: for Page:= 0 to 3 do   { 640 * 200 Pixel, 4 Speicherebenen }
           begin
             SetEReg(4,Page);  { Register 4: Read Map - zu lesende Ebene }
             if WriteScreen(16000,$400) <> 0 then Exit; { 16000 Bytes/Seite }
           end;
    EGAHi: for Page:= 0 to 3 do   { 640 * 350 Pixel, 4 Speicherebenen }
           begin
             SetEReg(4,Page);  { Register 4: Read Map - zu lesende Ebene }
             if WriteScreen(28000,$800) <> 0 then Exit; { 28000 Bytes/Seite }
           end;
    VGAHi: for Page := 0 to 3 do  { 640 * 480 Pixel, 4 Speicherebenen }
           begin
             SetEReg(4,Page);  { Register 4: Read Map - zu lesende Ebene }
             if WriteScreen(38400,$960) <> 0 then Exit; { 38400 Bytes/Seite }
             { In diesem Modus ist bei 256 KByte Video-RAM keine zweite
               Bildspeicherseite möglich. }
           end;
    EGAMonoHi:
           begin                    { 640 * 350 Pixel, 1 Speicherebene }
             SetEReg(4,0);      { Register 4: Read Map - zu lesende Ebene }
             if WriteScreen(28000,$800) <> 0 then Exit; { 28000 Bytes gesamt }
           end;
      end; { case }
    Close(f);
end;
```

SaveScreen **schreibt abhängig vom momentan gesetzten Videomodus zwischen 16000 und 38400 Bytes pro Bildspeicherseite auf die Diskette. Das Funktionsergebnis entspricht *IOResult*.**

```
function LoadScreen(name: string): Integer;  { = IOResult }
var f: File;  { untypisiert }
    Temp, Page: Integer;

function ReadScreen(Count,PageSize: Word): Integer;
{ Liest jeweils eine Speicherebene. Funktionsergebnis entspricht IOResult }
var Result: Word; ScrPnt: Pointer;
begin
  ScrPnt := Ptr($A000 + PageSize * ActivePage, 0);
  BlockRead(f,ScrPnt^,Count,Result);
  if Result <> Count then
  begin
    LoadScreen := 100;   { simuliert IOResult: Eof }
    ReadScreen := 100;
    Close(f);
  end
   else ReadScreen := 0;
end;
```

```
procedure SetMReg(Page: Integer);
{ Selektiert eine Leseebene über das Map-Register.
  Achtung: Die Portadressen sind hier nicht dieselben wie bei SetEGAReg! }
begin
  Port[$3C4] := 2;      { Selektiert Map Mask Reg }
  Port[$3C5] := Page;   { Bits 0..3 - Schreibebenen }
end;

begin  { LoadScreen }
  Assign(f,name); {$I-} Reset(f,1); {$I+}
  Temp := IOResult; LoadScreen := Temp;  { Fehler? }
  if Temp <> 0 then Exit;  { mit IOResult als Ergebnis }
  SetEReg(5,0);    { Mode-Register: Schreibebenen über Map Mask Reg }
  SetEReg(1,0);    { Enable Set/Reset Reg zurücksetzen }

  case GetGraphMode of
    EGALo: for Page:= 0 to 3 do     { 640 * 200 Pixel, 4 Speicherebenen }
           begin
             SetMReg(1 shl Page);  { Map Mask Register - Schreibebene }
             if ReadScreen(16000,$400) <> 0 then Exit; { 16000 Bytes/Seite }
           end;
    EGAHi: for Page:= 0 to 3 do     { 640 * 350 Pixel, 4 Speicherebenen }
           begin
             SetMReg(1 shl Page);  { Map Mask Register - Schreibebene }
             if ReadScreen(28000,$800) <> 0 then Exit;  { 28000 Bytes/Seite }
           end;
    VGAHi: for Page:= 0 to 3 do     { 640 * 480 Pixel, 4 Speicherebenen }
           begin
             SetMReg(1 shl Page);  { Map Mask Register - Schreibebene }
             if ReadScreen(38400,$960) <> 0 then Exit;  { 38400 Bytes/Seite }
             { In diesem Modus ist mit 256 KByte Video-RAM keine zweite
               Bildspeicherseite möglich. }
           end;
    EGAMonoHi: begin                  { 640 * 350 Pixel, 1 Speicherebene }
                 SetMReg(0);          { Map Mask Register - Schreibebene }
                 if ReadScreen(28000,$800) <> 0 then Exit;  { 28000 Bytes }
               end;
      end; { case }
    Close(f);
    SetMReg($F);       { erlaubt wieder alle vier Schreibebenen }
end;

end.   { keine Initialisierungen }
```

LoadScreen verläuft im Prinzip genauso wie *SaveScreen* und liefert ebenfalls *IOResult* als Funktionsergebnis zurück - nur daß hier die einzelnen Bildspeicherseiten nicht gelesen, sondern eben neu beschrieben werden.

Kapitel 6
Dateivariablen und Gerätetreiber

Vor langer, langer Zeit gab es einmal jemanden, der sein Pferd so oft morgens zur Arbeit schieben mußte, bis er schließlich den Unterschied zwischen *Driver* (= der Steuerer) und *Cowboy* (= Treiber) vergaß. Unglücklicherweise war dieser Mensch von Beruf Übersetzer. So wurde aus einem Steuerprogramm für angeschlossene Geräte (*device driver*) eben ein »Gerätetreiber«, von dem man bis heute nicht weiß, wen oder was er wohin treibt.

Mit dem Begriff »Dateivariable« könnte man in ähnlicher Weise herumalbern - nur daß diese irreführende Bezeichnung nicht auf einem Übersetzungsfehler, sondern auf den Grenzen der menschlichen Sprache beruht. Dateivariablen enthalten schließlich keine Datei, sondern eine Zustandsbeschreibung in Form von Namen, Zeigern und Indexen. Folgerichtig hat sie Niklaus Wirth in der Urdefinition von Pascal auch schlicht als *file pointer* (= Zeiger auf eine Datei) bezeichnet.

Die Evolution von Dateivariablen

Im einfachsten Fall (und in der ersten Definition von Pascal) besteht eine Dateivariable aus einem Zeiger, der mit einer Referenznummer des Betriebssystems verbunden ist. Das Lesen eines Zeichens aus einer (offenen) Datei geschieht mit zwei Pascal-Anweisungen*):

```
Ch := f^;  Get(f);
```

Die erste Anweisung setzt die Variable *Ch* mit dem Inhalt eines Puffers, der bei der Eröffnung der Datei gefüllt wurde; die zweite besteht aus einem Betriebssystem-Aufruf: »Lese das nächste Zeichen aus der Datei xxx in den Puffer«. Wie das Betriebssystem dieses »nächste Zeichen« liest, hat den Programmierer nicht zu interessieren - schließlich wurde Pascal nicht für einen bestimmten Computertyp definiert.

Eine Erweiterung durch die Version 3.0

Im Gegensatz zum Standard-Pascal ist die Version 3.0 alles andere als unabhängig vom Betriebssystem - sie leistet sich deshalb eine »öffentliche« Definition der internen Struktur von Dateivariablen. Der Begriff »Datei-Zeiger« erweitert sich damit auf einen ganzen Datenblock, der im Abschnitt 20.19.2.3.1 des 3.0-Handbuchs beschrieben und hier als (korrigiertes) Äquivalent wiedergegeben ist:

*) Die Form *Read(f,ch)* für Textdateien ist auch bereits im *User Manual and Report* definiert, wird dort aber als Abkürzung für den Programmierer bezeichnet - sozusagen als Makro für den Compiler.

```
type
  FileType = record
               Handle: Word;   { Referenznummer des Betriebssystems }
               case Boolean of
                  True:   (Flags, CharBuf: Byte);  { für Textdateien }
                  False:  (RecordLength: Word);   { für typisierte Dateien }
               BufPos:  Word;   { Index im Puffer }
               BufSize: Word;   { Puffergröße }
               BufAdr:  Word;   { Adresse des Puffers im Datensegment }
               BufEnd:  Word;   { momentanes Pufferende, nur für Input }
               Name:  array[0..63] of Char;  { Dateiname }
             end;
```

Dabei handelt es sich nach wie vor um eine echte Variable: Wie und wo der Puffer mit Daten aus der (durch *Handle* bezeichneten) Datei gefüllt bzw. Pufferinhalte in die Datei geschrieben werden, entzieht sich der Kontrolle des Programmierers. Anders gesagt: Mit Dateivariablen der Version 3.0 lassen sich ausschließlich Diskettendateien*) bearbeiten, weil der Compiler die dazugehörigen Lese- und Schreibroutinen automatisch und unabänderlich festlegt.

Dementsprechend eingeschränkt sind die Möglichkeiten, die eigentlich über ein »Beobachten« dieser Vorgänge nicht hinausgehen (wenn man einmal von direkten Manipulationen des Pufferinhalts absieht).

... und eine zweite durch die Version 4.0

Wie in Kapitel 23 des Referenzhandbuchs beschrieben, hat eine Text-Dateivariable seit der Version 4.0 die folgende Struktur:

```
TextBuf = array[0..127] of Char;   { Textpuffer }
TextRec = record
            Handle:    Word;        { DOS-Handle (Referenznummer) }
            Mode:      Word;        { Zugrifffsart }
            BufSize:   Word;        { Puffergröße }
            Private:   Word;        { von DOS reserviert }
            BufPos:    Word;        { Position (Index) im Puffer }
            BufEnd:    Word;        { Zahl momentan gespeicherter Zeichen }
            BufPtr:    ^TextBuf;    { zeigt auf den Puffer }
            OpenFunc:  Pointer;     { zeigt auf die Open-Routine }
            InOutFunc: Pointer;     { zeigt auf die Ein-/Ausgaberoutine }
            FlushFunc: Pointer;     { zeigt auf die Flush-Routine }
            CloseFunc: Pointer;     { zeigt auf die Close-Routine }
            UserData:  array[1..16] of Byte;  { beliebig verwendbar }
            Name:      array[0..79] of Char;  { Dateiname }
            Buffer:    TextBuf;     { Standard-Textpuffer }
          end;
```

*) Der Compiler behandelt *Input*, *Output*, *Lst*, *Aux* usw. ebenfalls als Diskettendatei - eine Unterscheidung wie zwischen »Schreiben auf die Diskette« und »Ausgabe über die parallele Schnittstelle« wird von DOS getroffen, das für derartige Operationen bestimmte *Handles* (1..5) reserviert.

Welche Routinen den Datenverkehr zwischen einer Dateivariablen und einer Diskettendatei, der Tastatur oder einer Zusatzkarte abwickeln, wird hier nicht mehr vom Compiler festgelegt, sondern durch den (variablen) Inhalt der Zeiger *OpenFunc..CloseFunc*. Mit diesem Konstrukt haben die Programmierer von Borland so etwas wie eine universelle Zwischenstation für Daten jeder Art geschaffen:

In der Versionen 4.0 und 5.0 dient eine Dateivariable nicht nur als Puffer zwischen dem Programm und einer Datei, sondern definiert auch die Art der Kommunikation - legt also fest, *wer* ihren Puffer füllt oder schreibt[1]. Ob eine über den Zeiger *InOutFunc* aufgerufene Schreibroutine tatsächlich DOS-Aufrufe zum Schreiben des Pufferinhalts auf die Diskette durchführt, eine Zusatzkarte anspricht oder eine Schnittstelle aktiviert, ist Sache des Programmierers.

»Treiber«-Routinen müssen...

eine ganze Reihe von Anforderungen erfüllen und haben ganz bestimmte Charakteristika. Damit die nächsten Seiten nicht mit »Vorsichts« und »Achtungs« gespickt sind, kommen die wichtigen Punkte[2] hier *en bloc*:

Jede der vier Routinen

- wird mit CALL FAR aufgerufen und muß deshalb explizit (mit {$F+}) als **far** deklariert sein, wenn sie nicht gerade im Interfaceteil eines Units verzeichnet ist (und deshalb automatisch als **far** codiert wird);
- bekommt bei ihrem Aufruf die Dateivariable übergeben und muß deshalb einen **var**-Parameter des Typs *TextRec* (oder eines Äquivalents davon) deklarieren, weitere Parameter sind weder zulässig noch sinnvoll;
- muß als Funktion mit dem Ergebnistyp *Integer* deklariert sein. Das Funktionsergebnis wird nach dem Aufruf als *IOResult* gesetzt, sollte also bei fehlerfreier Ausführung den Wert 0 haben bzw. im Fehlerfall einem der Standard-Fehlercodes entsprechen.

Da der Aufruf über Zeiger erfolgt, hat der Compiler keine Möglichkeit zur Prüfung, ob der Programmierer eine oder mehrere dieser Regeln verletzt.

Wie in Kapitel 25 des Referenzhandbuchs beschrieben, muß die »Zuordnungs-Prozedur« (*Assign...*) zumindest den Zeiger *OpenFunc* setzen - die restlichen Initialisierungen kann man der *Open*-Routine überlassen.

---

[1] Womit wir bei den zu Anfang des Kapitels beschriebenen Schwierigkeiten angelangt wären: Für eine Variable, die an der Festlegung ihres eigenen Inhalts beteiligt ist, existiert noch keine vernünftige Bezeichnung. Ich fühle mich allerdings auch nicht dazu berufen, hier ad hoc eine zu erfinden.

[2] Diese Liste ist nicht vollständig - sie enthält hauptsächlich Dinge, die mir im Referenzhandbuch nicht deutlich genug hervorgehoben erschienen.

Die *Open*-Routine muß die folgenden Felder setzen (soweit das noch nicht bei der Zuordnung geschehen ist):

- *BufPtr* auf den Puffer der Variablen, *BufPos* auf 0 (= Puffer leer), *BufSize* auf die Größe des Puffers;
- *InOutFunc*, *FlushFunc* und *CloseFunc* auf die entsprechenden Routinen.

Das Feld *Mode* sollte zur Sicherheit bei der Zuordnung den Wert *fmClosed* erhalten und wird durch die vier Treiber-Routinen nicht verändert. Prüfungen und weitere Veränderungen dieses Feldes erfolgen durch die Routinen der Laufzeitbibliothek.

Druckerausgaben im Eiltempo

Jeder kennt die wunderschönen Bilder aus der Werbung für Druckerpuffer: Neben einem Computer nadelt ein Drucker gemächlich vor sich hin, davor sitzt ein entnervter Büromensch und zählt die Ringe unter seinen Augen. Korrekterweise ist der arme Mann immer mit einem Textverarbeitungsprogramm beschäftigt - bei der Ausgabe von Grafiken wartet nämlich nicht der Computer, sondern der Drucker*).

Tatsache ist: Selbst mein AT mit seinem 286er-BIOS braucht zur Ausgabe von 64 KByte in einer Schleife ohne Sinn, Zweck und jede Art von Fehlerprüfung, also etwa

```
0100 31C9        XOR   CX,CX
0102 B84100      MOV   AX,0041          ; AH:= 0, AL := 'A';
0105 CD17        INT   17               ; Ausgabe
0107 E2F9        LOOP  0102             ; 65535 Wiederholungen
```

rund 10 Sekunden. Richtig langsam wird die Sache allerdings erst, wenn man keinen BIOS-Aufruf benutzt, sondern das DOS-Gerät LPT1 - wie es das Unit *Printer* tut:

```
uses Printer;
var x:Word;
begin
  for x:= 0 to 65535 do
     Write(Lst,'A');
end.
```

Dieses Prograämmchen ist (auf demselben AT) erst nach einer guten halben Minute fertig, die sich aus 10 Sekunden für den BIOS-Aufruf und rund 20 Sekunden für interne DOS-Operationen zusammensetzen. Der Zeitbedarf des Pascal-Programms selbst ist demgegenüber vernachlässigbar: wie sich mit diversen üblen Tricks (hauptsächlich: INT $21 = IRET) feststellen läßt, liegt er bei einer halben Sekunde.

*) Die zweite Erkenntnis, nachdem man sich endlich einen Druckerpuffer zugelegt hat: Jetzt warten Drucker und Puffer auch bei Textausgaben darauf, daß der Computer endlich zu Rande kommt...

Was den Entwicklern von Turbo Pascal aus Gründen der »Allgemeinkompatibilität« nicht möglich war, läßt sich für den Privatgebrauch recht einfach nachholen: ein Unit, das wie *Printer* mit einer Dateivariablen *Lst* arbeitet, alle vier möglichen parallelen Schnittstellen bedient, *IOResult* korrekt setzt, ohne Maschinenbefehle auskommt - und dabei schneller als der nackte BIOS-Aufruf arbeitet (8,2 Sekunden für 64 KByte, also mehr als dreimal so schnell wie die Ausgabe über *Printer*). Das Listing:

```
Unit FastList;  { 1/88, 12/88 as }
{ Ausgabe über die vier möglichen parallelen Drucker-
  Schnittstellen mit direktem Port-Zugriff }

{$S-,I-,R-}   { keine Prüfung von Stack, I/O und Bereichen }
Interface
uses Dos;   { wg. Registers, TextBuf }

var Lst: Text;  { wird über den Initialisierungsteil LPT1 zugeordnet }

procedure AssignPrt(var F:Text; LptNo: Byte);  { LptNo von 1..4 }
```

FastList deklariert nur zwei »öffentliche« Bezeichner: die Dateivariable *Lst*, die durch den Initialisierungsteil der ersten Druckerschnittstelle zugeordnet wird, und eine Prozedur *AssignPrt*, die beliebigen Text-Dateivariablen eine der vier möglichen Drucker-Schnittstellen zuordnet.

```
Implementation

type
  { TextBuf = array[0..127] of Char; ist im Unit DOS definiert }
  TextRec = record  { wird hier redeklariert, leicht verändert }
     Handle: Word;  { DOS-Handle }
     Mode  : Word;  { fmInput, fmOutPut usw. }
     BufSize: Word; { Größe des Textpuffers }
     Private: Word;
     BufPos: Word;  { Position innerhalb des Puffers }
     BufEnd: Word;  { Anzahl momentan gespeicherter Zeichen / nur Input }
     BufPtr: ^TextBuf;  { zeigt auf den Textpuffer }
     OpenFunc,
     InOutFunc,
     FlushFunc,
     CloseFunc: Pointer;  { zeigen auf die entsprechenden Routinen }
     PortNo, PortAddr: Word;  { zwei zusätzlich eingefügte Felder }
     UserFill: array[1..12] of Byte; { ist um 4 Bytes kleiner als normal }
     Name: array[0..79] of Char;
     Buffer: TextBuf;     { der Standard-Puffer daselbst }
   end;
```

Der Implementationsteil deklariert eine Version von *TextRec* mit zwei zusätzlichen Feldern (aber derselben Gesamtgröße). Wie die Bezeichnungen vermuten lassen,

speichert *PortNo* die Nummer der Drucker-Schnittstelle (1..4), *PortAddr* enthält die I/O-Adresse der Schnittstelle und wird durch die *Open*-Funktion gesetzt.

Im Prinzip ist eine eigene Version von *TextRec* nicht notwendig - technisch gesehen erfüllt das Original denselben Zweck, wenn man die ersten drei oder vier Bytes des Feldes *UserData* zur Speicherung dieser beiden Werte benutzt.

```
{$F+ ******* !!!! ******}
function PrtOpen(var F: TextRec): Integer;
var Regs: Registers;
begin
  F.PortAddr := MemW[$40:8 + F.PortNo shl 1]; { LPT1:$40:08, LPT2:$40:0A ... }
  if F.PortAddr = 0 then  { ==> nicht vorhanden }
  begin
    PrtOpen := 151;   { IOResult: "Unkown Unit" }
    Exit;
  end;
  with Regs do
  begin
    AH := 1;  { Subfunktion 1: Initialisierung }
    DX := F.PortNo;
    Intr($17, Regs);
    if (AH and $20) <> 0 then PrtOpen := 159  { IOResult für "Paper out" }
     else if (AH and $01) <> 0 then PrtOpen := 160  { "Time out" }
     else PrtOpen := 0;   { alles o.k. }
   end;  { with Regs }
end;  { PrtOpen }
```

Die *Open*-Funktion ermittelt die I/O-Adresse der über *PortNo* angegebenen Drukker-Schnittstelle, führt vorschriftsmäßig die Initialisierung über den BIOS-Interrupt $17 aus und prüft den dabei zurückgelieferten Statuscode.

Die Basisadressen der Parallel-Ports werden vom BIOS beim Start des Computers in das Segment $40 kopiert: auf $40:08 findet sich die Adresse von LPT1, auf $40:0A die Adresse von LPT2 usw. Einträge mit dem Wert 0 stehen für »Schnittstelle nicht installiert« - *PrtOpen* liefert in diesem Fall den Statuscode für »Unbekanntes/nicht vorhandenes Gerät« zurück.

Achtung: *OpenFunc..CloseFunc* sind Zeiger - und Aufrufe über Zeiger codiert der Compiler grundsätzlich als CALL FAR. Folglich müssen alle vier »Treiber« auch mit {$F+} als FAR deklariert sein.

```
function PrtIgnore(var F:TextRec): Integer;
{ "Dummy" }
begin
  PrtIgnore := 0;
end;
```

PrtIgnore tut überhaupt nichts und liefert lediglich das Ergebnis »fehlerfreie Ausführung« zurück. Diese Routine wird als *Close*-Funktion eingesetzt.

```
function PrtSendBuf(var F:TextRec): Integer;
{ Ausgabe des Dateipuffer-Inhalts }
const  TIMEOUTFACTOR = 182;   { Timeout nach maximal 10 Sekunden }
var
  P: Integer;          { Index im Puffer }
  IOP: Word;           { I/O-Adresse }
  TimeCount: LongInt;  { Zähler für "Timeout" }
  SysTick: LongInt absolute $0040:$006C;  { Clock Tick des Systems }
  Status: Byte;
begin
  IOP := F.PortAddr;           { I/O-Adresse des Ports }
  for P:= 0 to F.BufPos-1 do   { Schleife für den gesamten Pufferinhalt }
  begin
    TimeCount := SysTick;    { momentane Systemzeit }
    repeat
      Status := Port[IOP+1];     { Status: "ready"? }
    until (Status >= $80) or (SysTick > TimeCount + TIMEOUTFACTOR);
    if Status < $80 then
    begin
      PrtSendBuf:= 160;  { "Timeout" -> Geräte-Schreibfehler }
      Exit;
    end;
    Port[IOP]:= Byte(F.BufPtr^[P]);  { Zeichen ins Latch schreiben }
    if (Status and $20) <> 0 then
    begin
      PrtSendBuf:= 159;  { "Out of paper" }
      Exit;
    end;
    Port[IOP+2]:=$0D;    { geschriebenes Zeichen abschicken }
    Port[IOP+2]:=$0C;    { Strobe }
  end; { for P }

  PrtSendBuf := 0;       { alles o.k. }
  F.BufPos := 0;     { Puffer ist wieder leer }
end;
```

PrtSendBuf gibt den Inhalt des Puffers zeichenweise aus und spricht dabei die Drucker-Schnittstelle direkt an, wobei auf »Timeout« und »Papierende« geprüft wird. Wenn man von der via TIMEOUTFACTOR einstellbaren Zeitspanne bis zum endgültigen Abbruch und der dazugehörigen Zeitmessung über den »Clock Tick« absieht, ist diese Routine eine mehr oder weniger exakte Kopie der Drucker-Ansteuerung eines EPSON AX, eines IBM XT und eines Taiwan-Kompatiblen*).

Diese Routine wird durch *AssignPrt* nicht nur als *InOut*-, sondern auch als *Flush*-Funktion gesetzt.

*) Funktional ist die Ansteuerung in allen drei Computern dieselbe - deshalb habe ich mich nicht bis ins letzte Detail darum gekümmert, was dabei elektronisch geschieht.

```
{$F-}
procedure AssignPrt(var F:Text; LptNo: Byte);
begin
  with TextRec(F) do
  begin
    Handle   := $FFFF;          { ungültig }
    Mode     := fmClosed;       { Datei ist geschlossen }
    BufSize  := SizeOf(Buffer);
    BufPos   := 0;              { Puffer ist leer }
    BufPtr   := @Buffer;
    OpenFunc:= @PrtOpen;        { OPEN: sucht lediglich die I/O-Adresse }
    InOutFunc := @PrtSendBuf;   { gibt den Pufferinhalt aus }
    FlushFunc := @PrtSendBuf;   { dito }
    CloseFunc := @PrtIgnore;    { liefert nur das Ergebnis "o.k." }
    PortNo    := LptNo-1;
    Name[0]   := #0;
  end; { with TextRec }
end;
```

Die Prozedur *AssignPrt* setzt hier sämtliche Felder der Dateivariablen außer der I/O-Adresse der Schnittstelle. Wie im letzten Abschnitt erläutert, hätte man einen guten Teil dieser Zuweisungen auch in *PrtOpen* ausführen lassen können.

```
      { ********************************** }

begin    { Initialisierungsteil }

  AssignPrt(Lst,1); Rewrite(Lst);
end.
```

FastList hat (als erstes und einziges Unit dieses Buches) einen Initialisierungsteil, in dem die Variable *Lst* der Schnittstelle LPT1 zugeordnet wird. Der Aufruf von *Rewrite* führt letztendlich wieder in dieses Unit hinein, d.h. aktiviert die *Open*-Funktion.

Umleitungsmöglichkeiten für Ein- und Ausgaben

In den Versionen 4.0 bzw. 5.0 unterscheiden sich *Input* und *Output* in keiner Weise von anderen Text-Dateivariablen - abgesehen davon, daß sie durch den Initialisierungsteil von *System* vor dem eigentlichen Programmstart mit speziellen *Handles* geöffnet werden:

- *Input* erhält den *Handle* 0000. Bei Eingabe-Anforderungen über diese Datei liest DOS die Tastatur;
- *Output* erhält den *Handle* 0001. Ausgaben über diese Datei schreiben auf den Bildschirm - der »Standardausgabe« von DOS.

Das Einsetzen einer eigenen Treiber-Routine ist deshalb im Prinzip eine völlig problemlose Angelegenheit:

```
program NewIODemo;  { NEWIO.PAS }
uses Dos;  { nur wg. TextRec }

{$F+} function NewInOut(var F:TextRec): Integer; {$F-}
const ScreenBase = $B800;  { $B000 für Monochrom & Hercules }
var p: ^Char;
    x: Integer;
begin
  with F do
  begin
    for x := 0 to BufPos-1 do
    begin
       if x < 25 then p := Ptr(ScreenBase,x*160);
       p^:= BufPtr^[x];
    end;
    BufPos := 0;    { Puffer ist wieder entleert }
  end; { with F }
  NewInOut := 0;  { Ausführung war fehlerfrei }
end;

begin
  with TextRec(Output) do
  begin
    InOutFunc := @NewInOut;
    FlushFunc := @NewInOut;
  end;

  Writeln('Senkrecht!');
end.
```

In diesem Programm werden die Zeiger *InOutFunc* und *FlushFunc* der Variablen *Output* auf eine eigene Routine umgesetzt, die die Zeichen senkrecht untereinander in den Bildspeicher schreibt.

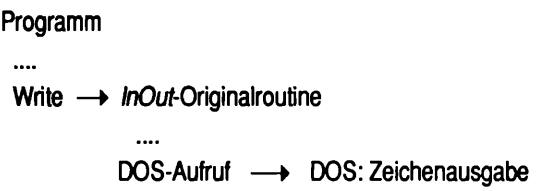
Eine erste Frage: Wieso müssen eigentlich *InOutFunc* und *FlushFunc* beide gesetzt werden? Und noch dazu auf dieselbe Routine? Weil sich die Funktionen *Flush* und *InOut* in einem Punkt unterscheiden: per Definition schreibt *Flush* den Puffer bei jedem Aufruf »leer«, während es *InOut* freisteht, Zeichen so lange zwischenzuspeichern, bis der Puffer wirklich randvoll ist.

Da *Write* und *Writeln* (ebenfalls per Definition) dafür sorgen, daß ausgegebene Zeichen auch physikalisch geschrieben werden, enden diese beiden Routinen mit einem Aufruf von *Flush*. Tatsächlich wird die *InOut*-Funktion überhaupt nur dann benutzt, wenn eine durchgehende Ausgabe mehr als 128 Zeichen erzeugt und deshalb ein zwischenzeitliches Schreiben des Pufferinhalts notwendig macht.

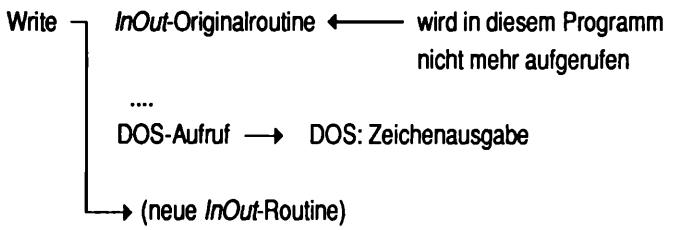
Die Laufzeitbibliothek leistet sich für *Ausgaberoutinen* nicht den Luxus einer Trennung zwischen *InOut* und *Flush*[1] - im Gegensatz zu *Eingaberoutinen*, bei denen die *InOut*-Funktion den Puffer füllt und die *Flush*-Funktion zum Überspringen bzw. Ignorieren gelesener Zeichen verwendet werden kann.

Eine zweite und durchaus ernstgemeinte Frage: Wieso gibt es keine Probleme mit dem Betriebssystem, wenn wir die Ausgaberoutine einfach ersetzen? Weil hier nur ein *Zeiger des laufenden Pascal-Programms* »verbogen« wird, der die Adresse einer Routine enthält, die schließlich einen DOS-Aufruf zur Zeichenausgabe durchführen würde. Grafisch ließe sich dieser Vorgang in etwa so darstellen:

Vor dem Ersetzen des Zeigers *InOutFunc*:



Nach dem Ersetzen des Zeigers:



Ein **Hinweis:** Die Anweisungsfolge *Assign(Output,'');* *Reset(Output);* baut die Dateivariable *Output* neu auf und setzt die Zeiger *OpenFunc..CloseFunc* wieder auf die Routinen der Laufzeitbibliothek.

Und schließlich die dritte und letzte Frage: Wie kriegt man nach dem Ersatz von *InOutFunc* und/oder *FlushFunc* überhaupt noch etwas auf den Bildschirm, ohne dabei im Bildspeicher herumstochern zu müssen? Zwei Möglichkeiten gibt es:

- über einen direkten DOS-Aufruf, wie ihn das letzte Programm dieses Kapitels (allerdings in anderem Zusammenhang) demonstriert;
- oder über eine Kette: der Zeiger wird auf eine Routine umgesetzt, die ihrerseits zu irgendeinem Zeitpunkt die originale *InOut*-Routine aufruft[2].

---

[1] Woraus folgt: das Vergrößern der Puffers mit *SetTextBufSize* bringt nur bei Leseoperationen Geschwindigkeitsvorteile. *Write* erzwingt das Leerschreiben eines 4000 Byte großen Puffers auch dann, wenn nur ein einziges Zeichen ausgegeben wurde.

[2] Wenn Sie jetzt auch nur für einen Moment gedacht haben, daß da etwas über *System.Writeln* oder *Crt.Writeln* zu machen wäre, dann sollten Sie sich schämen. Allerdings befinden Sie sich dabei in guter Gesellschaft mit dem Autor.

Ausgabeketten

Das folgende Programm fügt eine Routine mit eigenen Ansichten über den Buchstaben *E* »zwischen« der Variablen *Output* und der Ausgaberoutine ein:

```
program InOutLinkDemo;  { IOLINK.PAS 20-1-88 as }

uses Dos;  { nur wegen der Definition von TextRec }

var InOutFlush: Pointer;    { Adresse der Originalroutine }

function OrgIO (var OutP: TextRec): Integer;
inline ($FF/$1E/InOutFlush);  { Aufruf der Original-Ausgaberoutine }

{$F+} function NewInOut(var OutP: TextRec): Integer; {$F-}
var x: Integer;
begin
   with OutP do
   begin
     for x:= 0 to BufPos-1 do
        if Upcase(BufPtr^[x]) = 'E' then BufPtr^[x] := 'a';
   end;
   NewInOut:= OrgIO(OutP);
end;
```

Die eingeschobene Routine für *InOut* und *Flush* von *Output* ersetzt zuerst alle Vorkommen des Buchstabens *e* durch ein *a* und ruft dann die Original-Ausgaberoutine über die **inline**-Funktion *OrgIO* auf, wobei die Dateivariable als Parameter übergeben wird*).

```
procedure SetNewInOut;
begin
  with TextRec(Output) do
  begin
    if InOutFunc <> FlushFunc then
    begin
      Writeln('Getrennte Funktionen für InOut und Flush!');
      Halt(1);
    end;
    InOutFlush:= InOutFunc;   { Speichern des Originalwertes }

    FlushFunc := @NewInOut;
    InOutFunc := @NewInOut;
  end;  { with Output }
end;
```

*) Weitere Details zu Aufrufen über **inline** finden sich in den Kapiteln 1 und 3.

Die Installationsroutine prüft hier zuerst, ob die Zeiger auf *InOut* und *Flush* auch wirklich identisch sind - wenn ja, speichert sie die Original-Adresse und setzt danach *NewInOut* als Ausgaberoutine für *Output*.

```
  { ---------------------------------- }

var x: Integer;
begin   { Hauptprogramm }

   SetNewInOut;

  Writeln('Meine sehr geehrten Damen und Herren...');
  for x:= 1 to 100 do
     Write('aeEea');
  Writeln;
  Writeln('Numerische Werte bleiben unverändert: ', 44456);
end.
```

Auch wenn man's eigentlich weiß, was dabei herauskommt, sind die Auswirkungen dieser simplen Manipulation doch recht verblüffend.

AssignCrt und eigene Routinen

Der Initialisierungsteil von *Crt* führt ziemlich genau dieselben Schritte aus wie das erste Beispielprogramm: Er setzt die Zeiger *InOutFunc* und *FlushFunc* auf eine eigene Routine, die direkt in den Bildspeicher schreibt (eine Zwischenspeicherung des Originalwerts dieser beiden Zeiger findet dabei nicht statt).

Nach der Aufnahme von *Crt* sind im zuvor gezeigten Beispielprogramm *InOutLinkDemo* folglich keinerlei Modifikationen notwendig - der einzige Unterschied besteht darin, daß die Zeigervariable *InOutFlush* in diesem Fall nicht mehr eine Adresse von *System*, sondern eben die Adresse der *InOut*-Routine von *Crt* speichert.

Der Ersatz von *ConInPtr* und *ConOutPtr* in 3.0-Programmen

Diese beiden »3.0-Variablen« haben eine ähnliche Funktion wie die Felder *InOutFunc* und *FlushFunc* von Dateivariablen der Version 4.0 bzw. 5.0, sind aber ausschließlich auf die Eingabe via *Input* und die Ausgabe via *Output* begrenzt: die Zuweisung der Adresse einer Pascal-Routine an *ConOutPtr* legt die Ausgaberoutine für *Output* fest, analoges gilt für Zuweisungen an *ConInPtr* und *Input* bzw. die Spezialvariable *Kbd*.

Beide »Ersatz«-Routinen müssen zeichenweise und ohne Dateivariablen arbeiten:

```
procedure CharOut(Ch: Char); { sollte jeweils das Zeichen in Ch ausgeben }
 ....
ConOutPtr := Ofs(CharOut);  { folgende Ausgaben über Output geschehen
                              zeichenweise über die Routine CharOut }
```

```
function CharIn: Char;        { ersetzt die Leseroutine von Kbd }
  ...
ConInPtr := Ofs(CharIn);      { folgende Eingabeanforderungen über Input
                                rufen die Routine CharIn auf }
```

Diese beiden Variablen, die einem manchmal das Leben bei der Konvertierung existierender Programme schwermachen, lassen sich nachkonstruieren - nicht gerade einfach, dafür aber zu hundert Prozent:

```
program ConInOutPtr;  { CONIO.PAS, 16-1-88 as }
uses Dos;  { nur wg. TextRec und zur Demonstration }

var
  ConInPtr, ConOutPtr, ConOut40, ConIn40: Pointer;

procedure CallConOut(Ch:Char);
inline ($FF/$1E/ConOutPtr);  { Aufruf der 3.0-Ausgaberoutine }

function CallConOut40 (var OutP: TextRec): Integer;
inline ($FF/$1E/ConOut40);  { Aufruf der 4.0/5.0-Originalroutine }

{$F+} function ConOutLink (var F:TextRec): Integer; {$F-}
var x: Integer;
begin
  if ConOutPtr <> ConOut40 then    { wurde verändert: }
  begin
     with F do                     { zeichenweiser Aufruf }
     begin
       for x := 0 to BufPos-1 do
             CallConOut(BufPtr^[x]);
       BufPos := 0;    { Puffer ist wieder entleert }
     end; { with F }
     ConOutLink := 0;  { Ausführung war fehlerfrei }
  end
  else ConOutLink := CallConOut40(F);   { sonst Aufruf der 4.0/5.0-Routine }
end;
```

Die Funktion *ConOutLink* wird beim Start des Programms als *InOut*- und *Flush*-Routine von *Output* gesetzt. Wenn *ConOutPtr* nicht mehr auf die Original-Ausgaberoutine der 4.0- bzw. 5.0-Bibliothek zeigt (d.h. durch eine Zuweisung verändert wurde), wird die neue Routine in der Schleife aufgerufen und bekommt jeweils ein Zeichen übergeben. Falls *ConOutPtr* dagegen noch den ursprünglichen Wert hat, folgt ein Aufruf der Bibliotheksroutine mit der Dateivariablen als Parameter.

Da die Version 3.0 für Aufrufe über *ConOutPtr* kein Funktionsergebnis als Wert für *IOResult* definiert, liefert *ConOutLink* nach der Ausführung der Schleife immer das Ergebnis »fehlerfreie Ausführung« zurück.

```
function CallConIn :Char;
inline ($FF/$1E/ConInPtr);  { Aufruf der 3.0-Eingaberoutine }

function CallConIn40 (var OutP: TextRec): Integer;
inline ($FF/$1E/ConIn40);  { Aufruf der 4.0/5.0-Originalroutine }

{$F+} function ConInLink (var F:TextRec): Integer; {$F-}
begin
  if ConInPtr <> ConIn40 then  { wurde verändert: }
  begin
    with F do                  { 3.0-Routine }
     begin
       BufPtr^[BufEnd] := CallConIn;  { 1 Zeichen lesen }
       Inc(BufEnd);                  { danach BufEnd um eins erhöhen }
     end;
     ConInLink := 0;
  end
  else ConInLink := CallConIn40(F);  { sonst Aufruf der 4.0/5.0-Routine }
end;
```

Dasselbe für die Eingaberoutine: Sobald *ConInPtr* durch eine Zuweisung verändert wurde, folgt ein Aufruf der neuen Eingabefunktion, die keine Parameter erwartet und ein einzelnes Zeichen als Funktionsergebnis zurückliefert; ansonsten benutzt das Programm die Routine der Laufzeitbibliothek, der die Dateivariable übergeben wird.

```
procedure ConIOInit;
begin
  with TextRec(Output) do
  begin
    ConOutPtr := InOutFunc;   { Original-Ausgaberoutine }
    InOutFunc := @ConOutLink;
    FlushFunc := @ConOutLink;  { auf das "Bindeglied" }
  end;
  with TextRec(Input) do
  begin
    ConInPtr := InOutFunc;   { Original-Eingaberoutine }
    InOutFunc:= @ConInLink;
  end;
    ConOut40 := ConOutPtr;   { für die Prüfung auf "verändert" }
    ConIn40  := ConInPtr;
end;
```

Die Initialisierung: *ConOutLink* und *ConInLink* werden als »Bindeglieder« in *Output* bzw. *Input* eingesetzt, die Originalzeiger für den Vergleich doppelt gespeichert.

```
{ ********************** }

{$F+} procedure Out30(Chx:Char); {$F-}
var Regs: Registers;   { Chx als Unterscheidung zu Regs.CH }
begin
  with Regs do
  begin
    DL := Ord(Chx); AH := 2;
    MsDos(Regs);
 end;
end;

{$F+} function In30: Char; {$F-}
var Regs: Registers;
begin
  with Regs do
  begin
    AH := 1;
    MsDos(Regs);       { Zeichen lesen via DOS-Aufruf }
    In30 := Chr(AL);
  end;
end;
```

Zwei Beispielroutinen für Ein- und Ausgabe, die jeweils zeichenweise und ohne Dateivariable arbeiten.

Achtung: Diese Routinen werden über Zeiger aufgerufen und müssen deshalb als FAR codiert sein.

```
var s: string;
begin
  ConIOInit;
  Readln(s); Writeln(s);  { via 4.0/5.0 }

  ConInPtr := @In30; ConOutPtr:= Addr(Out30);
  Readln(s); Writeln; Writeln(s);  { via 3.0 }

end.
```

Ein kurzes Hauptprogramm, das die Zuweisungen an *ConInPtr* und *ConOutPtr* demonstriert - und ein letztes

Achtung: Die Zuweisungen müssen hier mit einer vollständigen Adresse stattfinden (d.h. mit @ oder *Addr* anstelle von *Ofs*).

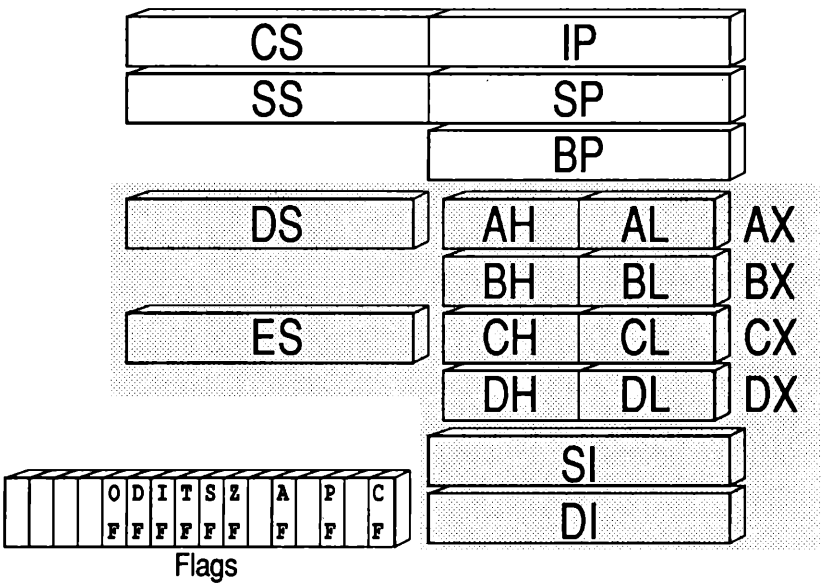
Anhang A
Programmiermodell des 8086

Die Computer der PC-Serie von IBM (und alle anderen »kompatiblen« MSDOS-Maschinen) verwenden einen Prozessor der Familie 80x86, der - je nach Ausführung - den Namen 8088, 8086, 80186, 80286 oder 80386 trägt. Der 8088 und der 8086 sind vom Standpunkt des Programmierers aus gesehen absolut gleich, die restlichen drei Versionen bieten zusätzliche Befehle und Betriebsarten, die von Turbo Pascal aber nicht genutzt werden. Auf die folgende Besprechung haben diese Erweiterungen absolut keinen Einfluß; wir werden deshalb im weiteren die Bezeichnung 8086 als Sammelbegriff verwenden.

Fangen wir mit einem kurzen Abriß des Prozessors aus der Sicht des Programmierers und einigen Allgemeinplätzen an:

Die Register des 8086

Das folgende Bild gibt die Register des 8086 aus der Sicht des Programmierers wieder.



Register des 8086

Was ist ein Register?

Der Prozessor verfügt »auf dem Chip« (d.h. innerhalb seines Plastikgehäuses) über einige RAM-Speicherzellen, die genauso wie der Hauptspeicher des Computers aufgebaut sind. Jede dieser Speicherzellen besteht also aus acht einzelnen Bits (»0« oder »1«) und kann Werte zwischen 0 und 255 (bzw. -128 .. +127) aufnehmen. Die meisten »Register« sind als zwei aufeinanderfolgende Speicherzellen (16 Bit, Werte von 0..65535 bzw. -32768..32767) organisiert.

Zu den Zellen des Hauptspeichers bestehen drei Unterschiede:

- Register sind nicht fortlaufend durchnumeriert - sie haben keine Adressen wie »Speicherzelle 0«, »Speicherzelle 1« usw., sondern werden über Namen wie AX, BX etc. angesprochen.
- Da die Register Teil des Prozessors sind, laufen Zugriffe mit wesentlich höherer Geschwindigkeit ab.
- Verschiedene Register sind (mehr oder weniger automatisch) miteinander verknüpft, d.h. ihre Inhalte beeinflussen sich gegenseitig.

Fast sämtliche Manipulationen von Daten laufen über die Register des Prozessors ab. Ein (leicht vereinfachtes) Beispiel für die Umsetzung einer Pascal-Anweisung in einzelne Prozessorbefehle:

```
var x: Integer;
begin
  x:= 0;              { 100: MOV AX,0 / MOV [x],AX            }
  repeat
   Inc(x);            { 101: MOV AX,[x] / INC AX / MOV [x],AX }
  until x = 10;       { 102: MOV AX,[x] / CMP AX,$000A        }
                      { 103: JNZ 101                          }
end.
```

Wenn wir davon ausgehen, daß die Prozessorbefehle (d.h. der Code des Programms) ab der Speicheradresse 100 stehen und für die Variable *x* zwei Bytes ab der Adresse 500 reserviert sind, dann wird das Register IP beim Start des Programms auf den Wert 100 gesetzt: Der Prozessor liest also den nächsten Befehl von dieser Adresse. Es ergeben sich die folgenden Schritte:

- Lesen des Befehls von der Adresse 100. IP wird dadurch automatisch um 1 erhöht und zeigt danach auf die Adresse 101. Die Ausführung des Befehls besteht aus zwei Schritten: »Lade das Register AX mit dem Wert 0« - »Speichere den Inhalt des Registers AX in den Speicherzellen 500 und 501«. Damit ist *x* auf den Wert 0 gesetzt.
- Lesen des Befehls von der Adresse 101. IP wird dadurch automatisch um 1 erhöht und zeigt danach auf die Adresse 102. Die Ausführung des Befehls besteht aus drei Schritten: »Lade das Register AX mit dem Inhalt der Speicherzellen 500/501« - »Erhöhe den Inhalt von AX um eins« - »Speichere den In-

halt von AX in den Speicherzellen 500/501«. Danach ist der Wert der Variablen *x* um eins erhöht.

- Lesen des Befehls von der Adresse 102. IP wird dadurch automatisch um 1 erhöht und zeigt danach auf die Adresse 103. Die Befehlsausführung besteht wiederum aus zwei Schritten: »Lade das Register AX mit dem Inhalt der Speicherzellen 500/501« - »Vergleiche, ob AX den Wert 10 enthält«. Dieser Vergleich setzt verschiedene Bits des Flag-Registers - dort wird also das Vergleichsergebnis (größer, kleiner, gleich usw.) festgehalten.
- Lesen des Befehls von der Adresse 103 (plus Erhöhung von IP um 1). Hier steht ein *bedingter Sprung*: »Springe zur Adresse 101, wenn das Flag-Bit 'ist gleich' nicht gesetzt ist«. Solange *x* also nicht den Wert 10 hat, wird ein Sprung zur Adresse 101 ausgeführt und der dort gespeicherte Befehl (Erhöhung von *x*) erneut bearbeitet. Woraus der »Sprung« besteht? Schlicht aus einer direkten Veränderung des Registers IP. In unserem Beispiel wird IP auf den Wert 101 gesetzt, der Prozessor liest also den nächsten Befehl wieder von dieser Adresse.
- Falls der vorherige Vergleich dagegen das Flag-Bit »ist gleich« gesetzt hat, wird der Sprungbefehl einfach ignoriert und das Programm mit dem Lesen des Befehls von der Adresse 104 fortgesetzt.

Beseitigen wir die Vereinfachungen:

1. Ein Prozessorbefehl kann zwischen ein und sechs aufeinanderfolgende Bytes umfassen. Erfreulicherweise erkennt der Prozessor am ersten Byte eines Befehls, welche Länge der Befehl insgesamt hat, und erhöht IP jeweils entsprechend.
2. Das »Setzen von IP auf 100« beim Start des Programms ist kein geheimnisvoller Mechanismus. MSDOS (ebenfalls ein ganz normales Programm, d.h. eine Folge von Prozessorbefehlen) lädt unser Beispielprogramm von der Diskette in den Hauptspeicher und führt dann einen Sprungbefehl (ohne vorangehende Bedingungen) aus: »Springe zur Adresse xxx« (d.h. setze IP auf den Wert xxx), wobei xxx für die Adresse steht, ab der das Programm in den Speicher geladen wurde.

Nach diesem Crash-Kurs können wir uns die einzelnen Register des Prozessors genauer ansehen:

Allzweck-Register

sind beim 8086 noch recht stark am traditionellen Aufbau eines Mikroprozessors orientiert. Eine ganze Reihe von Operationen läßt sich zwar mit jedem dieser Register durchführen, jedes Register hat aber seine eigenen und ganz speziellen Fähigkeiten. Zum Beispiel:

- AX ist das zentrale Register für arithmetische Operationen wie Addition, Subtraktion und Manipulation einzelner Bits mit AND, OR, XOR usw.
- CX arbeitet bei Schleifen als Zähler: Mit einem einzigen Befehl wird sein Wert um eins erniedrigt, danach folgt eine Prüfung, ob bereits der Wert 0 erreicht wurde, und abhängig davon ein Sprung.

Die Register BX und DX sind nicht so offensichtlich spezialisiert - es gibt aber auch einige Operationen, die nur über sie ausgeführt werden können (wie z.B. das Lesen und Beschreiben von I/O-Ports, bei dem DX die entsprechende I/O-Adresse festlegt). Der Austausch und Vergleich von Inhalten ist zwischen allen vier Registern in beiden Richtungen möglich (»Lade BX mit dem Inhalt von CX«, »Vergleiche, ob DX einen größeren Wert als CX enthält« usw).

Segmentregister

enthalten die *Segmentadressen* von Code, Daten und Stack. Sie lösen das Problem, wie man mit einem 16 Bit breiten Register (das Werte von 0..65535 enthalten kann), auf einen Gesamtbereich von einem Megabyte (ca. 1 Million verschiedener Speicherzellen) zugreift:

- jedem Lesen eines Befehls (d.h. jeder Adressierung via IP) wird der Inhalt des Registers CS vorangestellt;
- bei der Adressierung von Daten wird (normalerweise) über das Register DS gearbeitet;
- für Zugriffe auf den Stack ist das Register SS zuständig.

Das vierte Segmentregister (ES = Extrasegment) trägt seinen Namen zu Recht: es kann beliebig zur Adressierung von Daten verwendet werden.

Mit der Art und Weise, *wie* aus dem Inhalt eines Segmentregisters (16 Bit) und einer gegebenen Adresse (ebenfalls 16 Bit) eine vollständige Adresse mit 24 Bit gebildet wird, beschäftigt sich der übernächste Abschnitt.

Spezialregister

gibt es insgesamt sechs. Drei von ihnen (IP, SP und die Flags) werden automatisch gesetzt, die anderen (SI, DI und BP) verhalten sich im Prinzip wie Allzweck-Register, verfügen aber einerseits über ganz spezielle Möglichkeiten und sind andererseits in ihrer allgemeinen Verwendungsmöglichkeit etwas eingeschränkter - beispielsweise gibt es bei ihnen keine Unterteilungsmöglichkeit in zwei einzelne Bytes: (»SH« und »SL« als obere bzw. untere Hälfte von SI sind nicht definiert).

- Das Register SP (*stack pointer*) zeigt auf die momentane Spitze des Stacks (innerhalb des Stacksegments). Sein Wert wird durch Stack-Operationen (»push« und »pop«) automatisch verändert.

- Das Register BP (*base pointer*) ist eine Art »zweiter Stackzeiger« und wird für Indizierungen von Daten benutzt, die sich auf dem Stack befinden (übergebene Parameter bei Funktions- und Prozeduraufrufen).
- Die Register SI (*source index*) und DI (destination index) dienen einerseits als Quell- und Zielindex bei Kopien ganzer Speicherbereiche, andererseits werden sie von Turbo Pascal bei der Indizierung von Array-Elementen verwendet.
- Der Befehlszeiger (IP = *instruction pointer*) enthält die Adresse des jeweils nächsten zu lesenden Befehls (innerhalb des Codesegments). Er wird beim Lesen von Befehlen automatisch erhöht und durch Sprünge sowie Unterprogramm-Aufrufe verändert.
- Das Flag-Register ist das einzige Register des 8086 mit einer *bitweisen* Organisation: Hier werden die Ergebnisse von Vergleichen und anderen Operationen durch das automatische Setzen/Löschen entsprechender Bits (»Flaggen«) festgehalten.

Der Stack und Unterprogramme

Ohne die Tatsache groß zu erwähnen, haben wir die berüchtigten »Zeiger« bereits eingeführt: Das IP-Register enthält die *Adresse* des jeweils nächsten Befehls - und nicht etwa den Befehl selbst. Der Prozessor verfügt über ein zweites Register, das ausschließlich Adressen speichert: den Stackzeiger (SP). Die Funktion dieses Registers läßt sich am besten mit einem Pascal-Programm verdeutlichen:

```
program StackDemo_1;

const
  STACKSIZE = 2000;

var
  Stack: array[1..STACKSIZE] of Word;
  StackPtr: Integer;

procedure Push(w: Word);  { Legt den übergebenen Wert w auf den »Stack« }
begin
  Dec(StackPtr);          { Stackzeiger nach unten }
  Stack[StackPtr] := w;   { Eintrag }
end;

function Pop: Word; { Holt den obersten Wert vom »Stack« herunter }
begin
  Pop := Stack[StackPtr]; { Das ist der oberste Wert }
  Inc(StackPtr);          { Stackzeiger nach oben }
end;
```

```
begin
  StackPtr := STACKSIZE;
  Push(101);          { Wert 101 auf den »Stack« }
  Push(202);          { Wert 202 auf den »Stack« }
  Writeln(Pop);       { -> 202 }
  Writeln(Pop);       { -> 101 }
end.
```

Tatsächlich ist dieses Programm eine exakte Simulation des Prozessorstacks*):

- Die Variable *StackPtr* ist ein Index innerhalb eines bestimmten Speicherbereichs, der beim Programmstart auf das oberste Element initialisiert wird.
- Die Operation *Push* erniedrigt zuerst den Index innerhalb des Speicherbereichs und speichert den übergebenen Wert dann im entsprechenden Element.
- Die Operation *Pop* liefert den Wert als Ergebnis, auf den *StackPtr* innerhalb des Speicherbereichs zeigt, und erhöht *StackPtr* danach.

Woraus folgt: Solange wir auf das Array *Stack* nur über die Routinen *Push* und *Pop* zugreifen, erhalten wir die Elemente in der umgekehrten Reihenfolge ihrer Speicherung wieder zurück. Auf dieser Grundlage arbeiten alle Aufrufe von Unterprogrammen und die dazugehörigen Rücksprünge:

```
100    PROC1  CALL PROC2
101           ...

200    PROC2  MOV AX,0000
201           RET
```

Der Aufruf von PROC2 über den Befehl CALL besteht aus zwei Schritten:

1. Die Adresse des nächsten Befehls (»101«) wird als *Rücksprungadresse* auf den Stack gebracht, der Stackzeiger wird um zwei erniedrigt.
2. Danach wird IP auf die Adresse von PROC2 gesetzt (»200«): Der Prozessor liest also als nächstes den ersten Befehl von PROC2 (MOV AX,0000).

Der RET-Befehl am Ende von PROC2 liest die obersten zwei Bytes des Stacks in das IP-Register (also den Wert »101«) und erhöht SP wieder um 2. IP zeigt danach auf den Befehl CMP BX,0020 in PROC1 - die Programmausführung geht also wieder in PROC1 weiter.

Die folgende Erweiterung von PROC2 verdeutlicht den Ablauf beim Aufruf eines Unterprogramms, das seinerseits einen weiteren Unterprogramm-Aufruf ausführt:

*) Bis auf die Tatsache, daß das Prozessorregister SP nicht wort-, sondern byteweise arbeitet und bei »Push« und »Pop« jeweils um 2 erniedrigt bzw. erhöht wird.

```
200    PROC2  MOV AX,0000
201           CALL PROC3     ; bringt »202« auf den Stack
202           RET            ; »Pop« von »101«

300    PROC3  ....
301           RET            ; »Pop« von »202«
```

Der Stack läßt sich nicht nur für Rücksprungadressen, sondern auch für Parameter von Prozeduren und Funktionen verwenden. Eine zweite Version von *StackDemo*, die einen Prozeduraufruf mit der Übergabe zweier Parameter enthält:

```
program StackDemo_2;
                        { Der erste Teil ist unverändert }
const
  STACKSIZE = 2000;
var
  Stack: array[1..STACKSIZE] of Word;
  StackPtr: Integer;

procedure Push(w: Word);
begin   Dec(StackPtr);   Stack[StackPtr] := w;  end;
function Pop: Word;
begin   Pop := Stack[StackPtr];  Inc(StackPtr);  end;

{ ------------------------------- }
var x: Integer;

procedure Zwei_Parameter;  { (a,b: Word) }
begin
  x := Stack[StackPtr+1]     { Lesen des Parameters a }
     + Stack[StackPtr];      { Aufaddieren des Parameters b }

  Inc(StackPtr,2);     { Entfernen der Parameter vom Stack }
end;

begin
  StackPtr := STACKSIZE;
  Push(101);           { Parameter a auf den »Stack« }
  Push(202);           { Parameter b auf den »Stack« }
  Zwei_Parameter;      { Aufruf der Prozedur }
end.
```

Dieses Programm enthält eine erstaunliche Menge realistischer Einzelheiten:

- Vor dem Aufruf von *Zwei_Parameter* wird zuerst der Parameter *a*, danach der Parameter *b* auf den »Stack« gelegt.
- Innerhalb von *Zwei_Parameter* werden diese beiden Werte gelesen - und zwar nicht mit *Pop*, sondern durch einen direkten Zugriff über den momentanen

Wert von *StackPtr*. Der Parameter *a* wurde vom Hauptprogramm als erstes auf den Stack gelegt - er befindet sich deswegen *oberhalb* von *b*.

- Vor dem Rücksprung entfernt *Zwei_Parameter* die übergebenen Werte wieder vom Stack, indem *StackPtr* durch einen direkten Zugriff einfach erhöht wird.

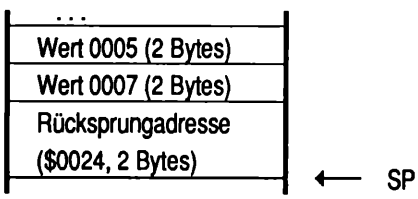
Sehen wir uns denselben Ablauf einmal *in vivo* an:

```
program StackDemo_Real;
var x: Integer;

procedure Zwei_Parameter(a,b: Word);
begin
  x := a + b;
end;

begin
  Zwei_Parameter(5,7);
end.
```

Der Aufruf von *Zwei_Parameter* wird vom Compiler in die folgenden Prozessorbefehle übersetzt:

```
** Zwei_Parameter(5,7);
0019 B80500        MOV    AX,0005            ; Wert 5 auf den Stack
001C 50            PUSH   AX                   (SP := SP - 2)
001D B80700        MOV    AX,0007            ; Wert 7 auf den Stack
0020 50            PUSH   AX                   (SP := SP - 2)
0021 E8DCFF        CALL   ZWEI_PARAMETER     ; Aufruf  (SP := SP-2)
0024       .....
```

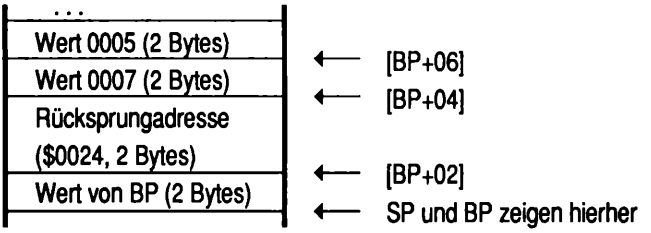
Der Stack vor der Ausführung des ersten Befehls von *Zwei_Parameter*:



Innerhalb von *Zwei_Parameter* wird zuerst das BP-Register auf dem Stack gespeichert (und SP damit erneut um zwei erniedrigt), danach bekommt BP den Wert von SP zugewiesen:

```
** procedure Zwei_Parameter(a, b: Word);
** begin
0000 55            PUSH   BP
0001 89E5          MOV    BP,SP
```

Der Stack sieht danach folgendermaßen aus:

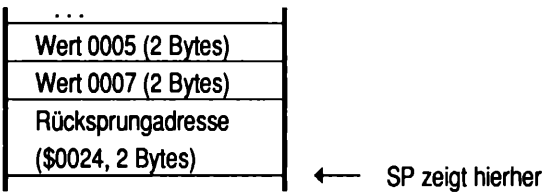


Wie in diesem Diagramm bereits angedeutet, lassen sich die Parameter *a* und *b* nun über BP (also sozusagen am Stackzeiger vorbei) adressieren.

```
** x := a + b;
0003 8B4606        MOV    AX,[BP+06]          ; Parameter a
0006 034604        ADD    AX,[BP+04]          ; + Parameter b
0009 A30000        MOV    [X],AX              ; -> x
** end;
000C 89EC          MOV    SP,BP
000E 5D            POP    BP
```

Das Ende von *Zwei_Parameter*: Die Kopie des Inhalts von BP in SP ist unnötig*), danach wird der ursprüngliche Wert von BP wieder vom Stack geholt.

Der Stack hat nun wieder denselben Inhalt wie zu Anfang der Prozedur:

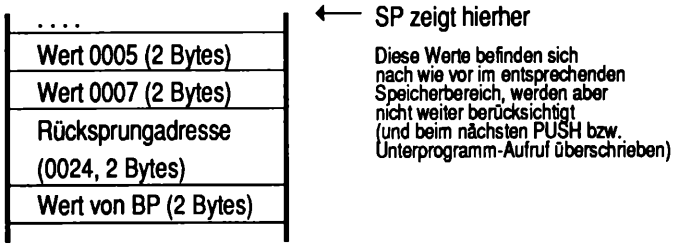


Schließlich kommt der Rücksprungbefehl, der in zwei Schritten ausgeführt wird: Zuerst liest der Prozessor die Rücksprungadresse in das IP-Register, danach wird SP um 4 erhöht:

```
000F C20400        RET    0004                ; a und b vom Stack
```

*) weil wir hier keine lokalen Variablen haben - siehe Kapitel 1.

Eine vollständige Darstellung des Stacks nach diesem Rücksprungbefehl:



Segmentierung des Speichers

Der 8086 verfügt über kein einziges Register, dessen Größe 16 Bit überschreitet (also Werte oberhalb von 65535 darstellen kann) - und trotzdem ist er in der Lage, auf insgesamt 1 048 576 verschiedene Speicheradressen zuzugreifen. Die Lösung des Rätsels liegt in der Unterteilung des Speichers in einzelne *Segmente* mit jeweils 64 KByte Umfang (also 65536 verschiedenen Adressen).

Die naheliegendste (und schlechteste) Lösung wäre eine Aufteilung in Schritten zu je 64 KByte (»banking«), wie sie aus den schlechten alten Zeiten von CP/M bekannt ist: Das erste Segment geht von 0..65535, das zweite von 65536..131071 usw. Der verfügbare Adressraum von 1 MByte wird also in 16 Blöcke mit jeweils 64 KByte Umfang unterteilt, wobei Anfangs- und Endadressen jedes Blocks unveränderlich festgelegt sind. Einer der augenfälligsten Nachteile dieser Methode: Ohne zusätzlichen Aufwand (Relozierung) müßte jedes Programm unabhängig von seiner tatsächlichen Größe einen eigenen Block belegen.

Um die Anfangs- und Endadressen von Segmenten so flexibel wie möglich zu halten, verwendet der 8086 eine Blockgröße von 16 Bytes und das folgende Schema zur Bildung einer vollständigen Adresse:

```
Segmentregister * 4 + 16-Bit-Adresse
```

Die angegebene 16-Bit-Adresse wird als *Offset-Anteil* (kürzer: »Offset«) bezeichnet - sie legt die Adresse *innerhalb* eines Segments fest. Durch die Multiplikation mit 16 ergeben sich die folgenden Startadressen der einzelnen Segmente:

Segmentregister	Startadresse
0	0
1	16
...	...
65534	1048544
65535	1048560

Bei der Verwendung hexadezimaler Notation dürfte der Zusammenhang etwas klarer werden:

Segmentregister	Startadresse
0	0
1	10
2	20
...	...
2A30	2A300
...	...
FFFE	FFFE0
FFFF	FFFF0

Offensichtlich läßt sich durch diese Multiplikation (technisch: ein Linksschieben um vier Bitpositionen, also um eine »Hex-Stelle«) der gesamte Adressbereich von einem Megabyte bearbeiten.

Nach der Berechnung der Segment-Startadresse wird der Offset (d.h. die angegebene 16-Bit-Adresse) einfach dazuaddiert, das Ergebnis ist eine vollständige Adresse mit 24 Bit. Zuerst ein einfaches, dann ein kompliziertes Beispiel:

```
Segment   Offset   Ergebnis
 4000      3D7F      40000
                   + 3D7F
                     43D7F

 2F84      0532      2F840
                   + 0582
                     2FD72
```

Dieses Konzept hat einige Vorteile gegenüber einer »linearen« Adressierung:

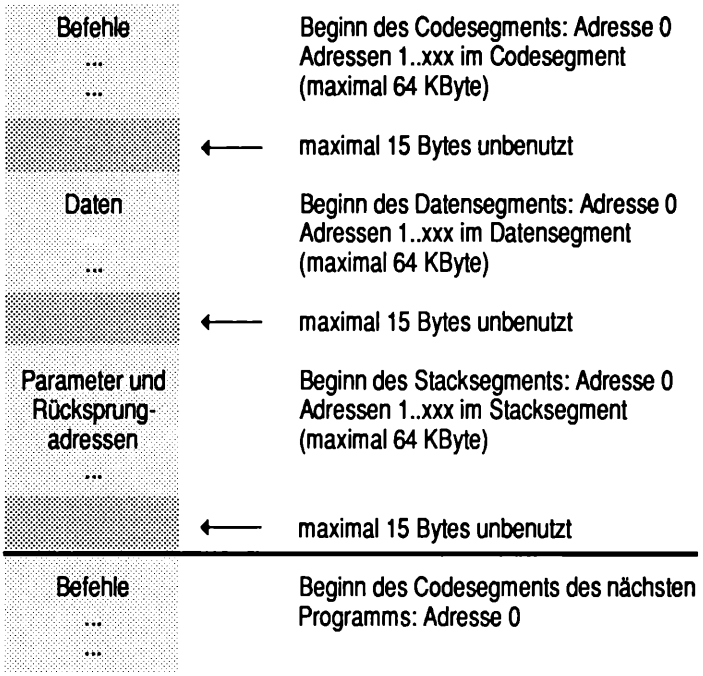
- Adressangaben können mit zwei Bytes codiert werden - der Prozessor spart sich das Lesen eines dritten Bytes, das bei einer vollständigen Adressangabe mit 24 Bit notwendig wäre. Programme werden durch diesen Kunstgriff nicht nur kürzer, sondern auch schneller.

- Der Aufwand zur Verwaltung des Speichers ist minimal, ein Relozieren von Programmen unnötig - jedes Programm kann ab der Adresse 0000 innerhalb eines Segments beginnen. Wenn der Speicher beispielsweise bis zur Adresse $2F7FF belegt ist, dann lädt DOS ein neues Programm ab der Adresse $2F800, setzt das CS-Register auf den Wert $2F80 und IP auf den Wert 0. Ohne die Unterteilung in Segmente müßte zu jedem Unterprogramm-Aufruf, Variablenbezug usw. die tatsächliche Startadresse des Programms (also $2F800) dazuaddiert werden.

- Da für Programmcode und Daten voneinander getrennte Segmentregister zur Verfügung stehen, gestaltet sich das »Reservieren von Speicherplatz« für Variablen ausgesprochen einfach: Der erste Befehl wird auf der Adresse 0000 im

Codesegment, die erste Variable auf der Adresse 0000 im Datensegment gespeichert. Anders gesagt: Die Speicheradresse der Variablen ist von der Länge des Programmcodes unabhängig.

Der verfügbare Hauptspeicher ließe sich also für mehrere Programme folgendermaßen unterteilen:



Wie groß die einzelnen Segmente tatsächlich sind, spielt keine Rolle - zwischen zwei Segmenten bleiben maximal 15 Bytes unbenutzter Raum. Wo sich das Programm tatsächlich im Speicher befindet, ist ebenfalls unerheblich, solange die Segmentregister für Code, Daten und Stack beim Start entsprechend gesetzt wurden.

Etwas Terminologie:

- *Jede* Adressierung verwendet ein Segmentregister (wenn man einmal vom Lesen und Beschreiben der I/O-Adressen absieht): Zum Lesen von Befehlen wird IP der Inhalt des CS-Registers vorangestellt; Adressierungen über den Stackzeiger geschehen grundsätzlich durch Voranstellen des Registers SS (Stacksegment). Für den Zugriff auf Daten wird standardmäßig das Register DS benutzt, durch entsprechende »Präfixe« können aber auch CS, SS und ES (»Extrasegment«) verwendet werden.
- Ein Segment muß wie oben demonstriert immer auf einer absoluten Adresse beginnen, die ein ganzzahliges Vielfaches von (dezimal) 16 darstellt. Diese

Unterteilung in Schritte zu je 16 Bytes wird als Unterteilung in *Paragraphs* bezeichnet - »xxx Paragraphs« steht also für »xxx Schritte mit je 16 Bytes«.

- Die Angabe vollständiger Adressen erfolgt hexadezimal und im Format *SSSS:OOOO*, wobei *SSSS* für die Segmentadresse und *OOOO* für den Offset-Anteil steht (also für die 16-Bit-Adresse innerhalb dieses Segments).

Genug des Jubels. Tatsache ist: Der 8086 wurde (genauso wie MSDOS) unter dem Gesichtspunkt maximaler Kompatibilität zum Betriebssystem CP/M konzipiert - er war bereits veraltet, als IBM die PC-Serie auf den Markt brachte.

(Ein Einschub, dessen einziger Zweck in der vorsorglichen Abwendung einer einstweiligen Verfügung des Herstellers Intel besteht: Der »protected mode« eröffnet den Prozessoren der 80x86-Familie mehr als zeitgemäße Möglichkeiten, und man darf gespannt sein, wie das Rennen zwischen den Konzepten 80x86, MC68xxx und NS32xxx ausgeht. Nur wird der »protected mode« von MSDOS nicht unterstützt und ist überhaupt erst ab dem 80286 möglich).

Für die Verwaltung von maximal 64 KByte Code, 64 KByte Daten und 64 KByte Stack ist der 8086 der ideale Prozessor - für alles, was darüber hinausgeht, ist er eine programmiertechnische Katastrophe.

Da Offsets auf einen Bereich von 64 KByte beschränkt sind, geraten Zugriffe auf Adressen »außerhalb des Programms« entsprechend kompliziert. Um beispielsweise den Wert der Speicherzelle 0000:0000 zu lesen, muß ein Programm die folgenden Schritte ausführen:

```
PUSH ES          ; Segmentregister zwischenspeichern
MOV AX,0000
MOV ES,AX        ; "MOV ES,0000" ist nicht definiert
MOV AL,ES:[0000] ; Lesen der Speicherzelle 0000:0000
POP ES           ; Segmentregister wieder zurück
```

Da ein Offset Werte von 0..65535 annehmen kann, die Startadressen der Segmente aber in Schritten von jeweils 16 Bytes variabel sind, läßt sich ein und dieselbe Adresse durch eine Vielzahl von Kombinationen darstellen:

```
0000:0123     absolut     0000:0123
0010:0023                 0000:0123
0012:0003                 0000:0123
0002:0103                 0000:0123
0008:00A3                 0000:0123
0006:00C3                 0000:0123
```

Im Prinzip spielt es überhaupt keine Rolle, mit welcher Kombination aus *Segment: Offset* eine absolute Adresse gebildet wird. Die Mehrdeutigkeit macht sich hauptsächlich bei der Arbeit mit Zeigern unangenehm bemerkbar: Ein direkter Vergleich wie

```
if p1 = p2 then ...   { wenn p1 und p2 auf dieselbe Adresse zeigen... }
```

würde auch dann *false* ergeben, wenn *p1* den Wert 1000:0123 und *p2* den Wert 1010:0023 enthalten, also beide auf dieselbe Adresse (nämlich 10123) zeigen. Damit derartige Vergleiche ein eindeutiges Ergebnis haben, hält man Zeigerwerte am besten in *normalisierter* Form - jeder Zeiger wird (über ein entsprechendes Unterprogramm) so umgerechnet, daß er einen Offset-Anteil von maximal $000F enthält:

```
var Temp: LongInt;
begin
  Temp := Segment_Anteil * 16 + Offset_Anteil;  { absolute Adresse }
  Segment_Anteil := Temp div 16;
  Offset_Anteil := Temp mod 16;
```

Aus einem Wert wie 1010:0023 wird in diesen Schritten also 1012:0003, die Umrechnung von 1000:0123 liefert dasselbe Ergebnis. (Obwohl sämtliche Programmiersprachen mit diesem Problem zu kämpfen haben und alle Prozessoren der Familie 80x86 über Multiplikationsbefehle verfügen, an denen mehrere 16-Bit-Register beteiligt sind, kennt erstaunlicherweise selbst der 80386 keinen Befehl, der einen in zwei Registern enthaltenen Zeigerwert in eine normalisierte Form bringen würde).

Aufrufe als near und far

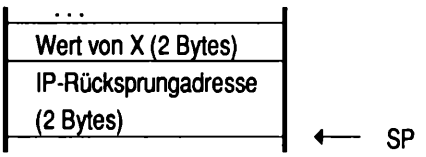
Sowohl beim Aufruf eines Unterprogramms (CALL) als auch beim abschließenden Rücksprung (RET) muß zwischen zwei Arten unterschieden werden:

- *near*-Aufruf und Rücksprung: Das Unterprogramm befindet sich im selben Codesegment. IP wird auf dem Stack gespeichert und belegt dort 2 Bytes, CS bleibt unverändert. Der RET-Befehl liest die obersten zwei Bytes des Stacks und setzt IP mit den gelesenen Werten. Da *near*-Aufrufe das CS-Register unverändert lassen, sind sie von der tatsächlichen Lage des Programms im Speicher unabhängig.
- *far*-Aufruf und Rücksprung: Das Unterprogramm befindet sich in einem anderen Codesegment. CS:IP (4 Bytes) werden auf dem Stack gespeichert und danach entsprechend neu gesetzt. Der RET-Befehl liest die obersten 4 Bytes des Stacks und setzt CS:IP mit den gelesenen Werten. *far*-Aufrufe arbeiten nicht mit relativen, sondern mit absoluten Adressen und müssen deshalb der Lage des Programms im Speicher angepaßt werden. (Hier liegt der hauptsächliche Unterschied zwischen den Dateitypen .COM und .EXE - .EXE-Dateien enthalten eine Tabelle mit Querverweisen auf alle absoluten Aufrufe und Sprünge, die von DOS gelesen und für entsprechende Anpassungen verwendet wird).

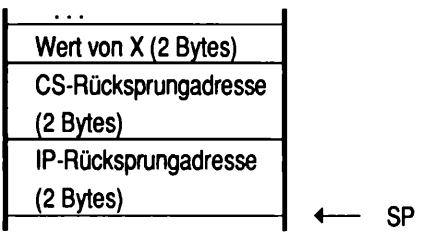
Der Prozessor hat keine Möglichkeit zur Prüfung, ob eine Routine als *near* oder *far* aufgerufen wurde - je nach Art des RET-Befehls liest er »bedingungslos« zwei bzw. vier Bytes vom Stack. Was passiert, wenn eine Routine als *near* aufgerufen wurde und mit einem RET FAR-Befehl endet? Nicht mehr viel - zumindest für den Benutzer des Programms: Der Prozessor liest zwei zusätzliche (und undefinierte) Bytes vom Stack in das CS-Register, und das Programm begibt sich in die ewigen Jagdgründe. Ähnlich verhält es sich mit übergebenen Parametern, deren relative Lage im Stack davon abhängt, ob die Routine mit zwei oder vier Bytes Rücksprungadresse aufgerufen wurde:

```
PUSH [X]     ; Wert von X auf den Stack
CALL PROC3
```

Wenn PROC3 als *near* aufgerufen wurde, dann sieht der Stack innerhalb der Routine folgendermaßen aus:



Der Wert von X befindet sich also 2 Bytes oberhalb von der Adresse, auf die SP zeigt. Wurde die Routine dagegen als *far* aufgerufen, dann enthält der Stack zusätzlich den Rücksprungwert für CS:



Der Wert von X befindet sich in diesem Fall 4 Bytes oberhalb von der Adresse, auf die SP zeigt*).

Interrupts und Vektoren

Der Begriff *Interrupt* (= Unterbrechung) steht eigentlich für ein (elektronisches) Signal, mit dem der Prozessor bei der Bearbeitung eines Programms unterbrochen

*) Diese Unterscheidung gehört so sehr zum täglich Brot des 8086-Programmierers, daß in Kapitel 1 gar nicht mehr groß darauf eingegangen wird.

werden kann. Ein gutes Beispiel für den Sinn dieser Möglichkeit ist die Art und Weise, wie die Computer der PC-Serie Eingaben über die Tastatur verarbeiten: Lassen Sie einmal einen längeren Text mit `type` auf den Bildschirm ausgeben und tippen Sie *während* der Ausgabe einige Zeichen ein - diese Zeichen erscheinen nach dem Ende der Ausgabe auf dem Bildschirm. Drei Möglichkeiten wären denkbar, wieso das funktioniert:

1. Die Tastatur speichert eingegebene Zeichen selbst und übergibt sie erst dann, wenn das System Zeit dafür hat. Voraussetzung dafür wäre, daß die Tastatur über einen eigenen Speicher verfügt.

2. Befehle wie `type` prüfen nach jeder Ausgabe, ob Zeichen eingegeben wurden. Diese Methode ist als *polling* bekannt und ausgesprochen unökonomisch: die Abfrage wäre in 99,9999% aller Fälle ergebnislos - der ständige Aufruf der entsprechenden Routine würde den Prozessor umsonst beschäftigen und von der Bearbeitung des eigentlichen Programms abhalten.

3. Die Tastatur löst ein Signal aus, das dem Prozessor kurzfristig eine Bearbeitungsroutine für Zeicheneingaben unterschiebt. Nach der Speicherung des Zeichens wird das Programm an der Stelle fortgesetzt, an der es zuvor unterbrochen wurde.

Die dritte Methode ist speziell dann die beste, wenn sich das System um eine Vielzahl von Peripheriegeräten (Uhr, Diskettenlaufwerke, Tastatur, serielle Schnittstellen usw.) kümmern muß: Es wäre recht teuer, jedes dieser Geräte mit einem zusätzlichen Puffer auszurüsten (Methode 1) bzw. ausgesprochen zeitaufwendig, ständig eine größere Zahl von Schnittstellen abzufragen (Methode 2).

Bleibt die Frage zu klären, wie der Prozessor zur *Behandlungsroutine* für einen Interrupt kommt. Weil diese Erklärung ziemlich weit in die Elektronik hineinführt, greifen wir erst einmal auf ein älteres Modell zurück:

1. Auf ein elektronisches Signal hin (d.h. die Änderung der Spannung an einem der Pins des Prozessorgehäuses) speichert der Prozessor den Befehlszähler und sein Flag-Register auf dem Stack.

2. Danach wird ein (fest vorprogrammierter) Sprung zu einer Adresse ausgeführt, ab der die Behandlungsroutine für Interrupts im Speicher stehen muß (oder ein weiterer Sprung, der zur eigentlichen Behandlungsroutine führt).

3. Die Behandlungsroutine muß alle Register vor Veränderungen zwischenspeichern (normalerweise wird sie dazu den Stack benutzen).

4. Danach kann sie die möglichen Quellen für den Interrupt abfragen und entsprechende Schritte zur Verarbeitung unternehmen.

5. Nach der Behandlung des Interrupts muß die Behandlungsroutine alle Register wieder auf die ursprünglichen Werte setzen und einen speziellen RET-Befehl ausführen, der auch das Flag-Register wieder vom Stack holt.

Diese Folge von Schritten gilt im Prinzip auch für den 8086 und die Elektronik des IBM PC - lediglich die Erkennung, welche Quelle den Interrupt ausgelöst hat, ist anders gelöst: Tastatur, Diskettenlaufwerke usw. sind nicht direkt mit dem Prozessor verbunden, sondern über *Interrupt-Controller*, der nun seinerseits zuerst ein Signal und danach ein Datenbyte (8 Bit) an den Prozessor übergibt. Die Interrupt-Controller sind programmierbar - beim Start des Systems wird jeder möglichen Quelle von Interrupts ein eigenes Datenbyte zugeordnet (z.B. $08 für die Uhr, $F1 für den Coprozessor usw).

Der 8086*) reserviert die untersten 1024 Bytes des Speichers für 256 *Vektoren* mit jeweils 32 Bit (also *far*-Adressen). Ein durch die eingebaute Uhr ausgelöster Interrupt läuft folgendermaßen ab:

1. Der Uhrenchip gibt ein Signal an den Interrupt-Controller ab. Der Interrupt-Controller gibt dieses Signal zusammen mit dem Datenbyte $08 an den Prozessor weiter.

2. Der Prozessor speichert das Registerpaar CS:IP zusammen mit den Flags auf dem Stack (Reihenfolge von oben nach unten: Flags, IP, CS) und liest das vom Controller übergebene Datenbyte ($08).

3. Der Prozessor führt einen Sprung zu der Adresse aus, auf die der Vektor $08 zeigt (sprich: er setzt CS:IP mit dem Inhalt der Speicherzellen 0000:$0020...0000:$0023).

4. Die Behandlungsroutine für den »Clock Tick« ist normalerweise ein Teil von DOS. Sie speichert die Prozessor-Register auf den Stack, die sie selbst benötigt, bringt die Zähler für Datum und Uhrzeit auf den neuesten Stand, lädt danach die Register wieder zurück und endet mit dem Befehl IRET, der CS:IP und die Flags wieder vom Stack liest.

Drei Einzelheiten zum Schluß dieser Betrachtung:

- Interrupts können sowohl durch Hardware-Signale als auch durch den Prozessorbefehl INT ausgelöst werden. Im ersten Fall liefert der Interrupt-Controller die Nummer des Vektors, im zweiten Fall ist sie Teil des Befehls. Weitere Unterschiede bestehen nicht - die restlichen Schritte sind absolut dieselben.

- Der Prozessor verfügt über ein Flag, mit dem Interrupts zeitweilig gesperrt werden können. Nach dem Befehl CLI (»Clear Interrupts«) werden Hardware-

*) Ich habe neulich recht verblüfft gelesen, daß dieser Speicherbereich nicht durch die Architektur des 8086, sondern durch die Programmierung des Interrupt-Controllers beim Start des Systems festgelegt wird. Falls dem tatsächlich so sein sollte: Die Speicherung der Vektoren in den untersten 1024 Bytes ist anscheinend durch IBM so zementiert, daß sich auch die Autoren diverser Bücher über den 80(x)86 nicht weiter darum gekümmert haben. Lassen wir es also mit dieser Fußnote bewenden.

Interrupts - nicht aber Aufrufe über INT - solange unterdrückt, bis das Flag mit dem Befehl STI (»Start Interrupts«) wieder zurückgesetzt wird*).

- Durch Veränderung der Vektoren (d.h. der Speicherzellen-Inhalte im Bereich von $0000:0000 bis $0000:03FF) läßt sich festlegen, zu welcher Adresse der Prozessor bei einem bestimmten Interrupt springt. Speziell bei Interrupts, die hardwaremäßig (und zyklisch) ausgelöst werden, sollte dazu die Routine *SetIntVec* des Units *Dos* verwendet werden - sie sorgt durch ein zeitweiliges Unterdrücken von Interrupts dafür, daß der entsprechende Vektor nicht ausgerechnet dann benutzt wird, während ihn das Programm gerade verändert.

*) Dieses Flag wird durch Interrupts erst einmal gesetzt. Genaugenommen ist also das erste CLI vor dem Umsetzen des Stacks in INTFRAME, DOSHOOK usw. unnötig.

Anhang B
Einige Hinweise zu MASM

Unabhängig davon, ob man über das zu MASM gehörige Handbuch verfügt oder das Programm anderweitig erworben hat - bei den ersten Versuchen beeindruckt dieser Makro-Assembler weniger mit seinen (fast unendlichen) Möglichkeiten, sondern eher mit einem gewissen Beharrungsvermögen auf bestimmten Formalitäten.

Oder auch: Was für Assembler-Programme mit mehreren Code-Segmenten, einigen hundert Prozeduren und einem eigenen Stack als Sicherung und Strukturierung eingebaut wurde, kann einen manchmal zur Verzweiflung treiben, wenn man eine halbes Dutzend Maschinenbefehle codieren will.

Dieser Anhang unternimmt nicht einmal ansatzweise den Versuch einer systematischen oder gar vollständigen Beschreibung von MASM - er beschränkt sich ausschließlich auf grundlegende Dinge, die im direkten Zusammenhang mit .OBJ-Dateien für Turbo Pascal-Programme stehen*).

Der Aufruf von MASM...

kann auf drei Arten erfolgen:

```
MASM Dateiname
```

Das Programm fragt einzeln nach den Namen für .OBJ-, .LST- und .CRF-Dateien. Das Suffix .ASM wird automatisch an *Dateiname* angehängt.

```
MASM Dateiname;
```

Das Programm übersetzt *Dateiname*.ASM ohne Rückfrage in *Dateiname*.OBJ, die Erzeugung eines Listings (.LST) und einer Kreuzreferenz (.CRF) wird unterdrückt.

```
MASM Dateiname,,,,
```

Das Programm übersetzt *Dateiname*.ASM ohne Rückfrage in *Dateiname*.OBJ und erzeugt dabei ein Listing (*Dateiname*.LST) sowie eine Kreuzreferenz (*Dateiname*.CRF).

Im Zusammenhang mit Turbo Pascal gibt es eigentlich nur einen speziellen Kommandozeilen-Parameter von MASM zu berücksichtigen: beim Aufruf mit

```
MASM /E Dateiname,,,,
```

werden 8087-Befehle »Emulator-gerecht« als Interrupts codiert (vgl. Anhang C).

*) Sprich: CLASS, GROUP, Makros, eigene Stacksegmente, LINK usw. usf. fallen unter den Tisch.

Programmcode

Das folgende Listing zeigt, welche Formalitäten für die Codierung von zwei Prozessorbefehlen im absoluten Minimalfall notwendig sind:

```
; MINI.ASM
CODE    Segment Public Byte   ; (1)
        Public MyProc         ; (2)
        ASSUME cs: CODE       ; (3)
MyProc  proc near             ; (4)
        mov ax,0              ; (5)
        ret                   ; (6)
MyProc  endp                  ; (7)
CODE    ends                  ; (8)
        end                   ; (9)
```

Einige allgemeine Dinge:

- Kommentare können an beliebiger Stelle einer Zeile stehen. MASM ignoriert sämtliche Zeichen, die sich zwischen einem Semikolon und dem nächsten Zeilenvorschub befinden.
- MASM unterscheidet normalerweise nicht zwischen Groß- und Kleinschreibung.
- Leerzeichen behandelt MASM wie der Compiler von Turbo Pascal: ein oder mehrere Leerzeichen werden als Trennung zwischen zwei Elementen erkannt und ansonsten ignoriert. Anstelle von Leerzeichen sind auch Tabs verwendbar.
- Jede Anweisung muß auf einer eigenen Zeile stehen und innerhalb dieser Zeile abgeschlossen werden - Zeilenvorschübe werden hier also auf andere Weise behandelt als in Pascal.
- Deklarationen von Bezeichnern (wie CODE und *MyProc*) müssen mit dem ersten Zeichen einer Zeile beginnen.

Die einzelnen Anweisungen in der Reihenfolge ihrer Zeilennummern:

1. Sämtliche Anweisungen, die in irgendeiner Form Prozessorbefehle oder Daten erzeugen, müssen sich innerhalb eines Segments befinden, d.h. von einer Deklaration der Form *Name* `Segment` und *Name* `ends` »eingerahmt« sein. .OBJ-Dateien, die in Turbo Pascal-Programme aufgenommen werden sollen, dürfen nur ein einziges Codesegment deklarieren, das den Namen CODE haben muß. (Dabei handelt es sich um eine willkürliche Festlegung: MASM zieht aus dem Namen CODE keine weiteren Rückschlüsse - in einem reinen Maschinenprogramm könnte dieses Segment auch MEIER heißen).

   Der Zusatz `Public` legt fest, daß der Bezeichner CODE auch außerhalb des Maschinenprogramms (d.h. für den Linker von Turbo Pascal) verfügbar ist.

Der Zusatz `Byte` legt fest, daß die folgenden Befehlsbytes auf beliebigen Speicheradressen beginnen können (und nicht explizit vom Linker auf einen Segment-Anfang geschoben werden müssen).

2. Legt fest, daß die folgende Prozedur *MyProc* auch außerhalb des Maschinenprogramms zur Verfügung steht - ansonsten könnte man *MyProc* nicht von einem Turbo Pascal-Programm aus aufrufen.

3. »Nimm an, daß das CS-Register bei den folgenden Prozessorbefehlen auf das Segment CODE zeigt«. MASM unternimmt hier *keine* weiteren Schritte (wie z.B. das Einfügen zusätzlicher Befehle). Diese Anweisung scheint hauptsächlich eine Konsequenz der Adressierung von Daten zu sein (siehe nächster Abschnitt).

4. Legt drei Dinge fest:

   ❑ Ab hier beginnt die Routine *MyProc*. Alle Routinen werden von MASM als `proc` bezeichnet - auch wenn ihr Code einer Pascal-*Funktion* entspricht.

   ❑ Aufrufe (CALL MYPROC) und Sprünge (JMP MYPROC) werden als *near* codiert. (Befehle wie CALL FAR *Name* weist MASM auch dann als fehlerhaft aus, wenn *Name* für eine als *far* deklarierte Routine steht).

   ❑ Alle RET-Befehle bis zum Abschluß von *MyProc* werden als *near* codiert (siehe Punkt 6).

5. Der erste Prozessorbefehl von *MyProc* (hier könnte eine beliebige Zahl weiterer Befehle folgen).

6. Ob dieser RET-Befehl von MASM als RET (NEAR) oder als RETF codiert wird, hängt einzig und allein von der Deklaration der Routine ab, in der er sich befindet*). Da wir uns innerhalb von *MyProc* befinden und diese Routine als *near* deklariert ist, codiert MASM den RET-Befehl als RET (NEAR).

7. Kennzeichnet das Ende der Prozedur *MyProc* (`endp` = *end procedure*). Auf diese Anweisung könnte die Deklaration einer weiteren Routine folgen, die sich dann ebenfalls im Segment CODE befindet.

8. Kennzeichnet das Ende des Segments CODE (`ends` = *end segment*). Auf diese Anweisung kann entweder die Deklaration eines weiteren Segments oder das Ende des Quelltextes folgen - etwas anderes ist nicht erlaubt.

9. Kennzeichnet das Ende des Quelltextes und hat eine ähnliche Wirkung wie das **end** eines Hauptprogramms in Turbo Pascal: eventuell noch folgender Text wird ignoriert.

---

*) Ein expliziter RETF-Befehl wird von MASM auch dann als Fehler ausgewiesen, wenn er in einer als *far* deklarierten Routine steht.

Deklarationen und *near*-Labels

MASM kennt nur ein Unterscheidungsmerkmal zwischen der Deklaration eines Bezeichners und einem Label (d.h. einem Sprungziel):

```
CODE    Segment Public Byte
        Public Proc1
        ASSUME cs: CODE
Proc1   proc near
        mov ax,0
L1:     call proc2          ; L1 ist ein near-Label
        call proc3
        jnz L1              ; Sprung zum Label L1
        ret
proc3:  dec ax              ; Proc3 ist NEAR und nur innerhalb
        ret                 ; des Maschinenprogramms verfügbar
Proc1   endp

Proc2   proc near           ; Proc2 ist eine eigene Prozedur, hier aber
L2:     dec ax              ; nicht als PUBLIC deklariert und deshalb
        ret                 ; nicht für Turbo Pascal verfügbar
Proc2   endp
CODE    ends
        end
```

Jede Angabe, die mit dem ersten Zeichen einer Zeile beginnt und mit einem Doppelpunkt endet, wird als *near*-Label betrachtet - alles andere dagegen als beliebige Deklaration*). *near*-Labels sind *keine* lokalen Bezeichner: sie dürfen innerhalb eines Segments nur einmal verwendet werden, ein Befehl wie `jmp L1` innerhalb von *Proc2* ist erlaubt, solange *Proc2* ebenfalls als *near* deklariert ist.

Ein **Hinweis:** Der Doppelpunkt wird nur bei der Erzeugung eines Labels angegeben. Bezüge auf Labels (JMP *Labelname*, CALL *Labelname* usw.) haben dieselbe Syntax wie Bezüge auf andere Bezeichner.

EXTRN-Deklarationen...

sind sozusagen das Gegenteil von PUBLIC - sie legen fest, welche Bezeichner das Maschinenprogramm »von außerhalb« erwartet. Für Routinen erwartet MASM außer dem Namen noch eine Angabe, ob das entsprechende Programmstück mit CALL (NEAR) oder CALL FAR aufgerufen werden soll:

```
CODE    Segment Public Byte
        Public Proc1
        EXTRN Proc2:near    ; wird mit CALL (near) aufgerufen
        EXTRN Proc3:far     ; wird mit CALL FAR aufgerufen
        ASSUME cs: CODE
```

*) Dementsprechend wunderliche Fehlermeldungen erhält man, wenn irgendwo ein Doppelpunkt vergessen wurde.

```
Proc1   proc near
        mov ax,0
        call proc2          ; von MASM als CALL codiert
        call proc3          ; von MASM als CALL FAR codiert
        ...
```

Wie bei einer Prozedur-Deklaration im Assembler-Quelltext bestimmt auch hier einzig und allein der Zusatz `near` bzw. `far`, ob MASM für Aufrufe und Sprünge *near*- oder *far*-Befehle erzeugt.

Datendeklarationen

Das folgende Beispiel demonstriert die grundlegenden Möglichkeiten, die dem Programmierer im Zusammenhang mit Turbo Pascal zur Verfügung stehen:

```
DATA    Segment Public Word
        EXTRN PasInt: Word      ; Pascal-Variable
Int1    dw ?                    ; darf NICHT initialisiert sein!
DATA    ends

CODE    Segment Public Byte
        Public Proc1
        ASSUME cs: CODE
        ASSUME ds: DATA
Int2    dw 222                  ; im Segment CODE - außerhalb der Prozedur
Proc1   proc near
        jmp L1
Int3    dw 333                  ; im Segment CODE - innerhalb der Prozedur

L1:     mov ax,Int1             ; Int1 im Segment DATA -> DS:Int1
        mov ax,Int2             ; Int2 im Segment CODE -> CS:Int2
        mov ax,Int3             ; Int3 im Segment CODE -> CS:Int3
        mov ax,PasInt           ; PasInt in DATA -> DS:PasInt

        ASSUME es:DATA, DS:nothing
        mov ax,Int1             ; Int1 im Segment DATA -> ES:Int1
        ret
Proc1   endp
CODE    ends
        end
```

Analog zum Segment CODE erlaubt der Linker von Turbo Pascal nur ein einziges Datensegment, das immer den Namen DATA haben muß. Daten können an vier verschiedenen Stellen deklariert werden:

- im Segment DATA als EXTRN, wobei MASM zusätzlich eine Größenangabe erwartet (siehe nächster Abschnitt);

- im Segment DATA als uninitialisierte Variable. Der Linker von Turbo Pascal reserviert entsprechend Platz im Datensegment, ignoriert aber jede Wertangabe;
- innerhalb des Segments CODE und außerhalb von Prozeduren;
- innerhalb einer Prozedur (die sich im Segment CODE befindet).

Wie das Listing zeigt, sind bei Datendeklarationen im Codesegment auch initialisierte Variablen möglich.

In diesem Zusammenhang bekommt auch die Anweisung ASSUME einen Sinn: Nach einem entsprechenden ASSUME »weiß« MASM, welches der Segmentregister momentan auf CODE bzw. DATA zeigt, und stellt automatisch das richtige Präfix voran, *erzeugt also aktiv zusätzlichen Code.*

Achtung: ASSUME-Anweisungen erzeugen keinen Code. Nach der Anweisung `ASSUME ds:DATA` zeigt das DS-Register also nicht plötzlich von selbst auf das Datensegment - ASSUME teilt dem Assembler lediglich mit, welche Segmentregister der Programmierer wohin gesetzt hat.

Wie der Compiler von Turbo Pascal nimmt MASM keine Analyse der Befehle vor:

```
        ASSUME cs:CODE, ds:DATA
Int3    dw 3333
        ....
        push cs
        pop ds        ; DS aufs Codesegment
        mov ax, Int3  ; <== MASM erzeugt den Befehl mov ax, CS:Int3
```

Hier wird also das CS-Register als Präfix vorangestellt, obwohl das DS-Register mittlerweile ebenfalls auf das Codesegment zeigt (und ein Präfix deshalb unnötig wäre). Ein zweites Beispiel:

```
DATA    Segment Public Word
        EXTRN PasInt: Word
DATA    ends
        ...
        ASSUME cs:CODE, ds:DATA, es:DATA
        ....
        push cs
        pop ds          ; DS aufs Codesegment
        mov ax, PasInt  ; <== MASM erzeugt den Befehl mov ax, PasInt, d.h.
        ....            ;     einen Zugriff auf das Segment CODE!
        ASSUME ds:CODE
        mov ax, PasInt  ; <== erzeugt den Befehl mov ax,ES:PasInt
```

Beim ersten Zugriff auf *PasInt* wurde MASM nicht mitgeteilt, daß das DS-Register nicht mehr auf das Datensegment zeigt - folglich wird es weiterhin verwendet (und der Programmierer hat sich die Folgen selbst zuzuschreiben). Beim zweiten Zugriff

auf *PasInt* weiß der Assembler dagegen, daß er das DS-Register nicht verwenden kann und stellt deshalb automatisch das Präfix ES voran.

Ein drittes Beispiel:

```
DATA    Segment Public Word
        EXTRN PasInt: Word
DATA    ends
        ...
        ASSUME cs:CODE, ds:DATA, es: nothing
        push cs
        pop ds          ; DS aufs Codesegment
        ASSUME ds:CODE
        mov ax, PasInt  ; <== erzeugt eine Fehlermeldung
```

Hier hat MASM festgestellt, daß momentan keines der Segmentregister auf das Datensegment zeigt, und erzeugt folglich eine Fehlermeldung*).

Eine ASSUME-Anweisung für ein Segmentregister ist für MASM solange gültig, bis sie durch ein weiteres ASSUME (für dasselbe Register) wieder aufgehoben wird - »Prozedurgrenzen« spielen dabei keine Rolle:

```
        ....
        ASSUME cs:CODE, ds:DATA
Proc1   proc far
        ....            ; <== keine weiteren ASSUMEs
Proc1   endp
Proc2   proc near
        ....            ; <== ASSUME cs:CODE, ds:DATA ist nach
        ....            ;     wie vor gültig
```

Datengrößen...

werden entweder durch die Angabe bei einer EXTRN-Deklaration, also

```
DATA    Segment Public Word
        EXTRN PasChar: Byte     ; ein Byte
        EXTRN PasInt: Word      ; ein Wort (2 Bytes)
        EXTRN PasLong: DWord    ; LongInt (4 Bytes)
        EXTRN PasSng: DWord     ; Single (Double Word, 4 Bytes)
        EXTRN PasDbl: QWord     ; Double (Quad Word, 8 Bytes)
        EXTRN PasExt: TByte     ; Extended (Tenbyte, 10 Bytes)
DATA    ends
```

*) Solange nicht eine entsprechende ASSUME-Anweisung gegeben wurde, nimmt MASM für jedes Register nothing an (die Vorgabe für ES ist in diesem Beispiel also nicht unbedingt notwendig).

oder durch ein zweibuchstabiges Kürzel bei der aktuellen Reservierung von Speicherplatz festgelegt:

```
DATA    Segment Public Word
aByte   db ?                ; reserviert ein (uninitialisiertes) Byte
aInt    dw ?                ; ein Wort
aTbyte  dt ?                ; reserviert 10 Bytes
DATA    ends

CODE    Segment Public Byte
aLong   dd 0445588AAh       ; Doppelwort (4 Bytes)
aSng    dd 3.5E+8           ; Single (4 Bytes)
aDbl    dq 8.238476E-23     ; Double (8 Bytes)
aExt    dt 8.4756E-2365     ; Extended (10 Bytes)

Table   dw 0055h, 9988h,0ABFh ; eine Tabelle mit drei Einträgen
MSG     db 'Eine Meldung'   ; eine Folge von Bytes
        ...
```

Explizite Größenbestimmung

MASM vergleicht bei jedem Datenzugriff die Größe von »Ziel« und »Quelle«. Unbedingt notwendig wäre eine solche Prüfung nur beim direkten Setzen von Speicherzellen wie bei

```
MOV [4857],77h
```

- weil hier aus dem Befehl nicht hervorgeht, ob ein einzelnes Byte ($77) oder ein Wort ($0077) gespeichert werden soll. Tatsächlich haben die Programmierer von Microsoft aber dafür gesorgt, daß sich der Assembler in *jedem* Fall mit einer Fehlermeldung beschwert, wenn bei einem Datentransfer die Größe von Ziel und Quelle nicht übereinstimmen:

```
DATA    Segment Public Word
aByte   db ?                ; reserviert ein (uninitialisiertes) Byte
aInt    dw ?                ; ein Wort
aTbyte  dt ?                ; reserviert 10 Bytes
DATA    ends
        ....
        ASSUME cs:CODE, ds:DATA
        ....
        mov al, aByte       ; o.k.
        mov ax, aInt        ; auch o.k.
        mov ax, aByte       ; *** Fehlermeldung
        mov al, aInt        ; *** Fehlermeldung
```

Der dritte Befehl stellt einen echten Fehler dar: hier würde der Prozessor den Wert von *aByte* und das niederwertige Byte von *aInt* in das AX-Register laden. Beim vierten Befehl sollte eigentlich klar sein, was der Programmierer will (nämlich das niederwertige Byte von *aInt*) - weil *aInt* aber als *Word* deklariert ist, erwartet

MASM eine explizite Angabe, daß auch wirklich nur ein Teil dieser Variablen geladen werden soll:

```
mov al, Byte Ptr aInt
```

Analoges gilt für alle anderen Teilzugriffe auf Daten:

```
mov Byte Ptr [xxxx], 0FFh   ; setzt ein einzelnes Byte
mov Word Ptr [xxxx], 0FFh   ; setzt xxxx auf $FF, xxxx+1 auf 0
mov ax, Word Ptr [aTByte+4] ; lädt die Bytes 04/05 der Variablen
                            ; aTByte in das Register AX
```

Daten vs. Adressen

Normalerweise verwandelt MASM jeden Datenzugriff in einen Lade- oder Speicherbefehl. Um ein Register nicht mit dem *Inhalt*, sondern mit der *Adresse* einer Variablen zu laden, muß deshalb der Zusatz OFFSET verwendet werden:

```
CODE    Segment Public Byte
        Public Proc1
        ASSUME cs: CODE
Msg     db 'Guten Tag!',0       ; ein ASCIIZ-String
Proc1   proc near
        push ds
        push cs
        pop ds                  ; DS aufs Codesegment
        ASSUME ds: CODE
        mov si,OFFSET Msg       ; Tabellenstart -> SI
L1:     lodsb                   ; erstes Byte von Msg
        ....
```

forwards jeder Art...

werden von MASM ausgesprochen kritisch betrachtet. Der Assembler erwartet bei Bezügen auf Bezeichner entweder, daß der entsprechende Bezeichner bereits deklariert ist (d.h. sich seine Deklaration vor dem Bezug im Quelltext befindet) - oder eine explizite Größenangabe. Das gilt sowohl für Daten als auch für Prozeduren:

```
CODE    Segment Public Byte
        Public Proc1
        ASSUME cs: CODE

Int2    dw 222
Proc1   proc near
        jmp L1
Int3    dw 333
```

```
L1:     push ds
        push cs
        pop ds                  ; DS aufs Codesegment
        ASSUME ds:CODE
        mov ax,Int2             ; problemlos
        mov ax,Int3             ; dito
        mov ax, Word Ptr Int4   ; explizite Angabe: Int4 ist ein Wort
        call Proc2              ; problemlos: Proc2 ist near
        call far Ptr Proc3      ; explizite Angabe: Proc3 ist far
        pop ds
        ret
Int4    dw 1234
Proc1   endp

Proc2   proc near
        ret
Proc2   endp

Proc3   proc far
        ret
Proc3   endp
CODE    ends
        end
```

Anhang C
Der 8087-Emulator im Detail

Daß die Grundidee des Emulators nur noch phantastisch zu nennen ist, wurde bereits in Kapitel 2 gesagt - umso befremdlicher scheint es, daß diesem Geniestreich bei Microsoft, Intel (oder wem auch immer) der Stempel »Streng geheim« aufgedrückt wurde. In der Dokumentation zu MASM 5.0 finden sich auf ziemlich genau 1000 Seiten zwei Hinweise:

- MASM erzeugt beim Aufruf mit dem Kommandozeilen-Parameter /E »Emulator-kompatible« 8087-Befehle (sprich: kennzeichnet die Opcodes als »relozierbar« und erweitert einzelne WAIT-Befehle auf WAIT/NOP);
- zusammen mit den Microsoft-Sprachen BASIC, Pascal, FORTRAN und C sind Emulator-Bibliotheken erhältlich.

Die Firma Borland, die das Prinzip des Emulators vermutlich*) in Lizenz erworben hat, macht da auch keine Ausnahme: Die amerikanische Ausgabe beschränkt sich auf die beruhigende Versicherung, daß die Sache a) vollautomatisch vonstatten geht, und b) schon funktionieren wird.

So weit, so schlecht. Erinnern wir die Rechtsabteilungen der genannten Firmen noch einmal an das Vorwort dieses Buches, in dem geschrieben steht, daß es sich bei allen Listings usw. um *Interpretationen* allgemein zugänglichen Materials handelt - und machen wir uns an die Arbeit.

8087-Opcodes und INT $3x-Befehle

Das Prinzip dieser wundersamen Wandlung wurde bereits in Kapitel 2 erläutert: grob gesagt, addiert der Compiler zu jeder Kombination *WAIT / 8087-Opcode* eine Konstante hinzu und macht so einen INT 3x-Befehl daraus. Woraus auch folgt: Um diesen INT 3x-Befehl wieder in sein Original zu verwandeln, muß die 8087-Laufzeitbibliothek dieselbe Konstante wieder abziehen - oder eine weitere Konstante hinzuaddieren, die zusammen mit der ersten genau 0x10000 ergibt (und den Überlauf unberücksichtigt lassen).

*) Die Gründe für diese Annahme: Der Emulator kann nur in paralleler Zusammenarbeit mit dem Hersteller des 8087 (Intel) entstanden sein; SYMDEB und CodeView sind meines Wissens die einzigen Debugger, die den Emulator direkt unterstützen (d.h. beim Listen eines mit Emulator-Interrupts durchsetzten Programms nicht hoffnungslosen Unsinn produzieren).

Wie man hinter diesen Mechanismus kommt? Man nehme ein Minimalprogramm wie

```
program Mult;
var X,Y,Z: Extended;
begin
  Z := X * Y;   { Ergebnis undefiniert, spielt aber keine Rolle }
end.
```

und lasse es im Modus {$N+,E-} übersetzen. Der erste 8087-Befehl des Hauptprogramms, mit SYMDEB vor seiner Ausführung (und nach dem Setzen des DS-Registers durch den Initialisierungsteil von *System*) gelistet:

```
000D CD372E4600     INT    37 ;FLD   TByte Ptr [Y]
```

Nach der Ausführung von *Mult* sieht dieser Befehl so aus:

```
000D 9B             WAIT1
000E DB2E4600       FLD    TByte Ptr [Y]
```

Wer nun nach dem Unterschied zwischen diesen beiden Befehlen sucht, einige andere 8087-Befehle auf dieselbe Weise analysiert (und/oder zufällig die Interna von Turbo C kennt), der endet schließlich bei 10 absoluten Konstanten, die hier der Übersichtlichkeit halber *en bloc* und zusammen mit ihren Namen wiedergegeben sind*):

```
0632  Abs  FISRQQ
0E32  Abs  FICRQQ
1632  Abs  FIERQQ
4000  Abs  FJARQQ
5C32  Abs  FIDRQQ
8000  Abs  FJSRQQ
A23D  Abs  FIWRQQ
C000  Abs  FJCRQQ
FE32  Abs  FIARQQ
```

Damit wäre die Materialsammlung fast beendet. (Wer meinen Forschungsweg komplett nachvollziehen will, der schreibe ein Programm X.ASM, das einen einzigen FLD-Befehl enthält, übersetze dieses Programm mit `MASM /e X;` und sehe sich dann an, welches FxxRQQ-Symbol beim Aufruf `LINK X;` als fehlend moniert wird. Für FLD ist es die Konstante FIDRQQ).

Der Rest wäre im Prinzip eine einfache hexadezimale Addition: LINK behandelt die Bytes

```
9BDB2E4600        (WAIT / FLD TByte Ptr [Y])
```

*) Wobei letztere aus dem 8087-Emulator von Turbo C stammen - Compiler und Linker von Turbo Pascal halten sich auch in diesem Punkt bedeckt. (Was uns noch zu einem anderen Punkt bringt: der 8087-Emulator von Turbo Pascal 5.0 ist in weiten Teilen mit dem der Turbo C-Version 2.0 identisch).

als Wert mit 16 Bit ($DB9B) und addiert die Konstante FIDRQQ ($5C32) dazu:

```
  DB9B
+ 5C32
  ----
= 137CD
```

Der Überlauf wird ignoriert, und das Ergebnis mit dem niederwertigen Byte zuerst gespeichert - aus $37CD wird also CD 37:

```
CD372E4600     INT    37 ;FLD   TByte Ptr [Y]
```

Der zur Rück-Umwandlung notwendige Wert ergibt sich aus der Berechnung $10000 - $5C32 (= $A3CE). Auch bei dieser Addition werden die Opcodes als Wert mit 16 Bit gelesen (d.h. mit $CD als niederwertigem Byte):

```
  37CD
+ A3CE
  ----
=  DB9B
```

Das Ergebnis - wieder als Folge von Opcodes geschrieben:

```
9BDB2E4600       (WAIT / FLD TByte Ptr [Y])
```

Ähnliches gilt für alle anderen FxxRQQ-Konstanten - auch wenn sich die Vor- und Rückwärtscodierung nicht immer mit einer einfachen Addition erledigen läßt, wie das Listing des Emulators im nächsten Abschnitt zeigen wird.

Aber Halt!

So geradlinig die Sache (trotz oder gerade wegen dieser verblüffenden Bit-Hexereien) auch aussieht: Der Teufel steckt wieder einmal im Detail. Tatsächlich funktioniert der Emulator *so* für Programme, die in Assembler, mit Turbo C, den Microsoft-Sprachen usw. geschrieben sind - kurz: für alles, was einen anständigen Linker sein eigen nennt. Auch wenn die Entwickler der Version 5.0 von Turbo Pascal bereits Microsoft-kompatible .MAP-Dateien beschert haben - zu einem »richtigen« Linker konnten sie sich offensichtlich doch nicht entschließen. Was bleibt, ist die Tatsache, daß sich nicht nur der Compiler, sondern auch der Linker vom Schalter {$E} reichlich unbeeindruckt zeigt: Wie bereits in Kapitel 2 entsetzt festgestellt, arbeiten Programme auch im Modus {$E-} mit Emulator-Interrupts, die erst zur Laufzeit in 8087-Opcodes umgewandelt werden. Auch reine 8087-Programmierer kommen deshalb um den nächsten Abschnitt nicht herum:

Der Dispatcher für INT $34..$3D

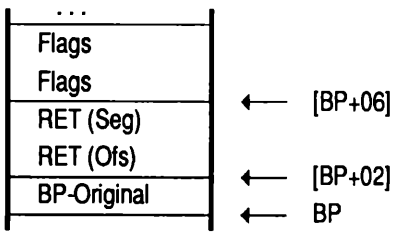
Wie in Kapitel 2 beschrieben, ist dieser Dispatcher in *jedem* Programm enthalten, das mit Fließkommazahlen arbeitet - unabhängig vom Stand des Schalters {$E}.

Die im Modus {$E+} eingebundene Version von INIT87 setzt diesen Dispatcher nur nach einer erfolgreichen Prüfung auf »Coprozessor vorhanden« in die Vektoren 0$34..$3D ein (und verwendet ansonsten den 8086-Emulator); im Modus {$E-} bricht INIT87 mit der Meldung »Coprocessor required« ab, falls das System nicht mit einem 80x87 ausgerüstet ist.

Ein letzter **Hinweis**, bevor es in die Details des Listings geht: da die Turbo Pascal-Variante des 8087-Emulators dasselbe Segment wie *System* benutzt, hängen die (Offset-)Adressen von Anzahl und Umfang der in das Programm aufgenommenen Standardroutinen ab.

```
******************************
* Dispatcher für INT $34..$3B *       Version 5.0
******************************
0A88 55            PUSH   BP
0A89 8BEC          MOV    BP,SP
0A8B 50            PUSH   AX                    ; AX, SI und DS
0A8C 56            PUSH   SI                      auf den Stack
0A8D 1E            PUSH   DS
0A8E F6460702      TEST   Byte Ptr [BP+07],02   ; INTs erlaubt?
0A92 7401          JZ     0A95
0A94 FB            STI                          ; ja, wieder anschalten

0A95 C57602        LDS    SI,[BP+02]            ; RET-Adresse
0A98 8B44FF        MOV    AX,[SI-01]            ; INT-Nummer&nächstes Byte
0A9B 2C34          SUB    AL,34                 ; INT $34->0, ... , $3B->7
0A9D 3C08          CMP    AL,08
0A9F 730C          JNB    0AAD
```

Ein Aufruf durch einen Interrupt speichert die Flags zusammen mit einer vollständigen RET-Adresse (*Segment:Offset*) auf dem Stack, der nach Ausführung der Befehle `PUSH BP` / `MOV BP, SP` folgendermaßen belegt ist (vgl. Kapitel 1):



Das Programm prüft mit einem Zugriff auf `[BP+07]`, ob das »Interrupt Enable«-Bit im Flag-Register *vor* dem Interrupt gesetzt war - wenn ja, werden Interrupts auch innerhalb des Emulators zugelassen.

Der Prozessor »weiß« natürlich nicht, daß der Compiler an den INT-Befehl noch einige weitere Bytes angehängt hat - er speichert bei einem Aufruf wie

```
(* 2000:000D CD372E4600      INT    37 ;FLD   TByte Ptr [Y]  *)
```

deshalb die Adresse 2000:000F für den Rücksprung*). Der Befehl `LDS SI, [BP+02]` lädt das Registerpaar DS:SI mit der Rücksprungadresse, `MOV AX, [SI-01]` lädt AX mit der Nummer des Interrupts und dem darauffolgenden Byte. Im gezeigten Beispiel kommt dabei der Wert $2E37 heraus - der Prozessor behandelt auch hier die Opcodes als Wort mit 16 Bit und interpretiert $37 als niederwertiges, $2E als höherwertiges Byte.

```
; ** INTs $34..3B - FIDRQQ: $34..$3B Opcl -> WAIT / Opcl
0AA1 83EE02         SUB    SI,+02                ; SI auf den INT-Befehl
0AA4 8104CEA3       ADD    Word Ptr [SI],A3CE    ; Umwandlung int WAIT/Opcl

- Xref: INT $3D (NOP / WAIT)
0AA8 897602         MOV    [BP+02],SI            ; RET-Adresse auf Op
0AAB EB23           JMP    0AD0                  ; Exit, Ausführung im Prog.
```

Die Rück-Umwandlung der 8087-Befehle, die über die Konstante FIDRQQ in Interrupts verwandelt wurden: SI wird um zwei erniedrigt und zeigt damit auf den INT 3x-Befehl selbst - für das zuvor gezeigte Beispiel also auf die Adresse 0100.

Wie im ersten Abschnitt dieses Anhangs demonstriert, kommt bei der Addition von $A3CE wieder die ursprüngliche Befehlsfolge (WAIT / FLD...) heraus, das dritte Befehlsbyte (im zuvor gezeigten Beispiel: $2E) bleibt unverändert.

Achtung: Bei der Addition werden die beiden Opcodes wieder als *Wort* behandelt - eine Bytefolge wie CD 37 (= INT 37) hat für den Prozessor also den Wert $37CD.

Der Befehl `MOV [BP+02],SI` setzt den Offset-Anteil der RET-Adresse auf den (»ehemaligen«) INT-Befehl (im Beispiel: auf 000D) - der Prozessor bekommt nach dem Rücksprung also die Befehlsbytes zu lesen, die soeben umgewandelt wurden.

```
0AAD 7726           JA     0AD5                  ; Sprung für INT <> $3C

0AAF F6C420         TEST   AH,20                 ; Bit 5 gesetzt?
0AB2 752E           JNZ    0AE2                    -> ja
```

Der Interrupt $3D (= WAIT / NOP) wird getrennt (ab $0AD5) bearbeitet, über den Interrupt $3C hatte Borland in Turbo C (Versionen 1.0 und 1.5) einige Zusatzfunktionen eingebaut, die durch ein gesetztes Bit 5 im dritten Byte gekennzeichnet sind. Diese Zusatzfunktionen sind in Turbo Pascal nicht definiert: der Programmteil ab $0AE2 holt die Register wieder vom Stack und springt zum laufenden Programm zurück.

*) Woraus auch folgt, daß der Emulator die RET-Adresse verändern *muß* - ansonsten würde der 8086 nach dem Rücksprung versuchen, die folgenden 8087-Operanden als Befehle zu interpretieren (und dabei entsprechenden Unsinn anstellen). Etwas kürzer: Wer einen EMU-Interrupt »kappt«, bringt sein Programm mit Sicherheit zum Absturz.

```
; ** INT $3C: FIARQQ Präfix DS ($3E)
              FICRQQ Präfix CS ($2E)
              FIERQQ Präfix ES ($26)
              FISRQQ Präfix SS ($36)
0AB4 83EE02        SUB    SI,+02                  ; SI auf INT-Befehl
0AB7 897602        MOV    [BP+02],SI                und als RET-Adresse
0ABA C6049B        MOV    Byte Ptr [SI],9B        ; WAIT voranstellen
0ABD 46            INC    SI
0ABE 8AC4          MOV    AL,AH                     Dx        Dx
0AC0 25C007        AND    AX,07C0                 ; 0000 0xxx 1100 0000
0AC3 D0E8          SHR    AL,1                      0000 0xxx 0110 0000
0AC5 D0E8          SHR    AL,1                      0000 0xxx 0011 0000
0AC7 D0E8          SHR    AL,1                      0000 0xxx 0001 1000
0AC9 3418          XOR    AL,18                     .... .... ...1 1...
0ACB 0526D8        ADD    AX,D826      26 = ES:     1101 1000 0010 0110
0ACE 8904          MOV    [SI],AX                   |ESC |01| Seg-Präfix
```

Lade- und Speicherbefehle des 8087, die mit Segment-Präfixen arbeiten, sind über den Interrupt $3C codiert. Hier ist die Rück-Umwandlung etwas komplizierter, weil der zu erzeugende 8087-Opcode nicht mit der sonst üblichen »ESC«-Bitfolge 11011 (= $D8+x) beginnt, sondern eben mit einem Segment-Präfix ($26/36 bzw. $2E/3E).

Sämtliche Informationen sind hier in dem Byte enthalten, das direkt hinter dem Interrupt-Befehl (CD 3C) steht. Aus

```
  CD  39    xx011yyy     <- 3. Byte bitweise interpretiert
|INT-Befehl|
```

wird

```
  9B  Seg-Präfix 11011yyy      <- 1. 8087-Opcode (D8..DF)
| WAIT |
```

(Falls Sie Vergnügen an Bit-Tricksereien haben: Die Bits 7 und 6 des 3. Bytes enthalten die Information, welches Segment-Präfix vorangestellt werden soll (11 = ES, 10 = CS, 01 = SS, 00 = DS), die von mir als *yyy* bezeichneten Bits stehen für das erste Operandenfeld des 8087 (das sich der »ESC«-Folge 11011 im 1. 8087-Opcode direkt anschließt), sie bleiben unverändert. Die drei Schiebeoperationen, das XOR $18 und die anschließende Addition von $26 (= `ES:`) dienen ausschließlich der Umrechnung von B7/B6 des dritten Bytes in ein Segment-Präfix).

```
0AD0 1F            POP    DS
0AD1 5E            POP    SI
0AD2 58            POP    AX                      ; DS, SI, AX und BP
0AD3 5D            POP    BP                        wieder vom Stack
0AD4 CF            IRET                           ; Ausführung im Programm
```

Das gemeinsame Ende aller 8087-Bearbeitungsfunktionen.

```
; ** INT $3D -> NOP / WAIT
0AD5 3C09          CMP    AL,09                ; > INT $3D?
0AD7 7709          JA     0AE2                 ; falls ja: Exit, ignoriert
0AD9 83EE02        SUB    SI,+02               ; SI zurück auf INT $3D-Op
0ADC C704909B      MOV    Word Ptr [SI],9B90   ; = NOP / WAIT
0AE0 EBC6          JMP    0AA8                 ; RET-Adresse setzen, Exit

0AE2 EBEC          JMP    0AD0                 ; Exit
```

Die Bearbeitung des Interrupts $3D, d.h. seine Umwandlung in die Befehlsfolge NOP / WAIT: Dieser Programmteil ignoriert »höhere« Interrupts als $3D[1] und überschreibt den INT-Befehl im aufrufenden Programm einfach direkt. Weiter geht es mit der Adresse $AD0 , wo der Wert von SI als RET-Adresse eingesetzt wird.

Der »Shortcut«-Interrupt $3E

Über diesen Interrupt haben die Entwickler von Turbo Pascal eine Reihe einfacher 8087-Operationen zusammengefaßt, die man »immer mal wieder braucht« und die zusammen mit zwei Zusatzbytes in der folgenden Form aufgerufen werden:

```
INT $3E <Subcode>/NOP   ; Subcode von $DC..$FE
```

Hier gibt es mehrere erstaunliche Dinge zu vermelden: zum einen erzeugt der Compiler selbst keine $3E-Interrupts - zum anderen ist es mir beim größeren Teil dieser »Abkürzungen« nicht gelungen, einen Verwendungszweck ausfindig zu machen[2]. Und noch etwas: SYMDEB und CodeView behandeln INT $3E-Befehle auf völlig normale Weise (d.h. ohne die Interpretation zusätzlicher Bytes oder eingefügte Kommentare) - was der Vermutung Raum gibt, daß es sich dabei um eine Eigenentwicklung von Borland handelt.

Die Adressen der einzelnen Subfunktionen folgen direkt auf den Dispatcher der Interrupts $34..$3D im Codesegment (wobei nicht verwendete Routinen mit einem Sternchen gekennzeichnet wurden):

```
       Start   Subcode     Funktion
       ------------------------------------------------------------
0AE4   0B29  *   DC    -   Entlädt die 8087-Register auf den 8086-Stack
       0B3D  *   DE    -   Lädt die 8087-Register vom 8086-Stack
       0B4E  *   E0    -   Vermutlich Prüfung der Form A <= X <= B
       0B53  *   E2    -   dito, aber ohne Abbruch bei INVALID
       0B8A  *   E4    -   Vergleich TOS/R1 mit zwei POPs
```

[1] Von woher sie auch kommen mögen - der Vektor $3E ist auf eine andere Adresse gesetzt.

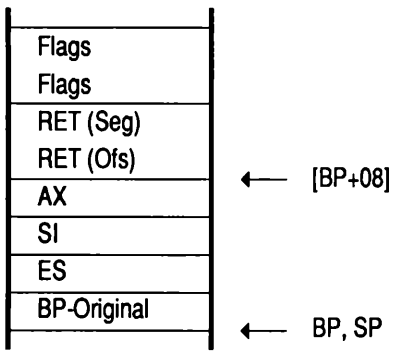
[2] Man compiliere ein sehr großes Programm (wie beispielsweise MCALC) und lasse dann die Bytefolge $CD $3E von DEBUG oder SYMDEB suchen. Das magere Ergebnis sind fünf Fundstellen im Codesegment von *System* (*Sin*, *Cos*, *ArcTan*, *Ln* und *Exp*).

```
0B9C  *  E6   -  Vergleich TOS/R1 mit POP
0BA6  *  E8   -  FTST (Prüfung des TOS-Wertes)
0BAB  *  EA   -  FXAM (Prüfung des TOS-Wertes)

07CF     EC   -  Sin
07D3     EE   -  Cos
07D7     F0   -  ArcTan
08C8  *  F2   -  Mehrweg-Prüfung des TOS-Inhalts
093B     F4   -  Ln (FLDLN2 unters TOS)
0931  *  F6   -  FLD1 unters TOS
0936     F8   -  FLDLG2 unters TOS
099B     FA   -  Exp (FLDL2E unters TOS)
09A2  *  FC   -  dieselbe Routine wie $09A6, aber ohne FLDL2T
09A6  *  FE   -  FLDL2T unters TOS
```

Darauf folgt der Dispatcher für $3E-Interrupts:

```
************************
* INT $3E ("Shortcut") *
************************
0B08 FB          STI
0B09 50          PUSH   AX
0B0A 56          PUSH   SI                  ; AX, SI, ES und BP
0B0B 06          PUSH   ES                    auf den Stack
0B0C 55          PUSH   BP
0B0D 8BEC        MOV    BP,SP
0B0F C47608      LES    SI,[BP+08]          ; IRET-Adresse -> ES:SI
0B12 26AD        LODSW  ES:                 ; nächstes Byte + NOP
0B14 897608      MOV    [BP+08],SI          ; IRET-Adresse plus 2
0B17 3CDC        CMP    AL,DC               ; Subcode < $DC?
0B19 7209        JB     0B24                ; -> ja, undefiniert: Exit
```

Die Ermittlung des Subcodes und die Erhöhung der Rücksprungadresse läuft hier nach einem ähnlichen Verfahren wie für die Interrupts $34..$3D. Der Inhalt des Stacks vor dem Zugriff mit `LES SI,[BP+08]`:



Der Befehl LODSW lädt die Rücksprungadresse in AX und erhöht SI gleichzeitig um zwei - danach zeigen ES:SI also auf die beiden Bytes, die im Programmcode unmittelbar auf den INT $3E-Befehl folgen.

```
0B1B 8BF0          MOV   SI,AX              ; Subcode
0B1D 98            CBW                      ; -> als (negatives) Wort
0B1E 96            XCHG  AX,SI              ; zurück in SI
0B1F 2EFF94080B    CALL  CS:[SI+0B08]       ; Aufruf der Subfunktion
```

Die Ermittlung der Subfunktions-Startadresse läuft hier etwas eigenwillig ab: der Subcode ($DC..FE) wird als negativer Wert auf 16 Bit erweitert - und damit beim Aufruf über `[SI+0B08]` von $0B08 *abgezogen.*

```
0B24 5D            POP   BP
0B25 07            POP   ES                 ; Register wieder vom
0B26 5E            POP   SI                   Stack und Exit (zur
0B27 58            POP   AX                   IRET-Adresse + 02)
0B28 CF            IRET
```

Mit diesem Programmteil enden alle via INT $3E aufgerufenen Funktionen.

Wie die Listings einiger $3E-Funktionen zeigen, läßt der Emulator - im Gegensatz zum Dispatcher der Interrupts $34..$3D - den aufrufenden Code auch dann unverändert, wenn ein Coprozessor im System installiert ist:

```
; INT $3E, Subfunktion $DC  (Unload der 8087-Register)
0B29 FECC          DEC   AH
0B2B 7C0E          JL    0B3B               ; AH = 0 -> Exit
0B2D B00A          MOV   AL,0A              ; 10 Bytes pro Register
0B2F F6E4          MUL   AH                 ; * Registerzahl
0B31 96            XCHG  AX,SI
0B32 9B            WAIT
0B33 DB7A0E        FSTP  TByte Ptr [BP+SI+0E]  ; Zielbereich: Stack!
0B36 83EE0A        SUB   SI,+0A             ; TOS -> SI + 10 * x
0B39 7DF7          JGE   0B32               ; R1  -> SI + 10 * x -1
0B3B 9B            WAIT                     ; R2  -> SI + 10 * x - 2
0B3C C3            RET                      ; ...

; INT $3E, Subfunktion $DE   (Laden der 8087-Register)
0B3D 2BF6          SUB   SI,SI              ; "MOV SI, 0000"
0B3F FECC          DEC   AH                 ; 0 Register?
0B41 7C09          JL    0B4C               ; -> ja, Exit
0B43 9B            WAIT
0B44 DB6A0E        FLD   TByte Ptr [BP+SI+0E]  ; Quellbereich: Stack!
0B47 83C60A        ADD   SI,+0A             ; Rx   <- SI + 0
0B4A EBF3          JMP   0B3F               ; Rx+1 <- SI + 10
0B4C 9B            WAIT                     ; Rx+2 <- SI + 20
0B4D C3            RET                      ; ....
```

Diese beiden Routinen könnten beispielsweise durch den Debugger bei der (programmunabhängigen) Berechnung von Ausdrücken verwendet werden (was sich aber nicht nachweisen ließ). Sie erwarten als einzige $3E-Funktionen neben gesetzten 8087-Registern noch einen zusätzlichen Parameter - die Anzahl der zu ladenden bzw. zu »entladenden« Werte.

Achtung: Vor dem Aufruf dieser Funktionen muß entsprechend Platz auf dem Stack reserviert worden sein - falls nicht, ist es um lokale Variablen und Rücksprungadressen der aufrufenden Routine geschehen. Ein (korrektes) Beispiel:

```
sub sp,4 * 10       ; 40 Bytes Platz (= 4 8087-Register)
mov ah,4            ; TOS bis R(3) sollen "entladen" werden
int 03Eh
db  0DCh, 90h       ; Subcode $DC / NOP
.....
<beliebige 8087-Operationen>
.....
mov ah,4            ; Zurückladen der 4 8087-Register vom Stack
int 03Eh
db  0DEh, 90h       ; Subcode $DE / NOP
add sp, 4 * 10      ; Stack zurück
....
```

SP muß hier natürlich beim Zurückladen der 8087-Register denselben Wert wie bei der Speicherung haben.

```
; ** INT $3E, Subcode $E0: Vermutl. Bereichsprüfung (mit Abbruch bei INVALID)
0B4E B000          MOV    AL,00                  ; IM nicht gesetzt
0B50 EB03          JMP    0B55
0B52 90            NOP

; ** INT $3E, Subcode $E2: Vermutl. Bereichsprüfung (ohne INVALID-Interrupt)
0B53 B001          MOV    AL,01                  ; IM gesetzt

0B55 55            PUSH   BP
0B56 50            PUSH   AX                     ; Platz fürs Kontrollwort
0B57 8BEC          MOV    BP,SP
0B59 9B            WAIT
0B5A D97E00        FSTCW  [BP+00]
0B5D 9B            WAIT
0B5E 8B7600        MOV    SI,[BP+00]           # ; Original-CW retten
0B61 806601FC      AND    Byte Ptr [BP+01],FC    ; DeNorm & Invalid Mask raus
0B65 084601        OR     [BP+01],AL             ; Subcode $E0: IM bleibt
0B68 9B            WAIT                            ungesetzt; $E2: gesetzt
0B69 D96E00        FLDCW  [BP+00]                ; Kontrollwort -> 8087
0B6C 9B            WAIT
0B6D D9EE          FLDZ                          ; 0
0B6F 9B            WAIT
0B70 DCEA          FSUB   ST(2),ST               ; minus R2
```

```
0B72 9B          WAIT
0B73 DEC1        FADDP  ST(1),ST          ; + R1
0B75 897600      MOV    [BP+00],SI      # ; Original-CW wieder setzen
0B78 9B          WAIT
0B79 D96E00      FLDCW  [BP+00]
0B7C 9B          WAIT
0B7D DED9        FCOMPP ST(1)             ; Vergleich
0B7F 9B          WAIT
0B80 DD7E00      FSTSW  [BP+00]           ; Ergebnis -> 8086-Stack
0B83 9B          WAIT
0B84 58          POP    AX                ; und von dort aus -> AX
0B85 5D          POP    BP
0B86 88660C      MOV    [BP+0C],AH
0B89 C3          RET
```

Wie bereits im Listing durch skeptische Kommentare angedeutet, sind Sinn und Zweck dieser beiden Routinen (mangels Aufrufbeispiel) nicht so ganz geklärt.

```
; ** INT $3E, Subcode $E4: Vergleich TOS/R1 mit doppeltem POP
0B8A 9B          WAIT
0B8B DED9        FCOMPP ST(1)

- Xref: Exit für eine Reihe kürzerer $3E-Routinen
0B8D 8A660D      MOV    AH,[BP+0D]
0B90 9B          WAIT
0B91 DD7E0C      FSTSW  [BP+0C]
0B94 9B          WAIT
0B95 8A460D      MOV    AL,[BP+0D]        ; Ergebnis (Flags)
0B98 89460C      MOV    [BP+0C],AX        ; (via Dispatcher -> AX)
0B9B C3          RET

; ** INT $3E, Subcode $E6: Vergleich TOS/R1 mit POP
0B9C 9B          WAIT
0B9D D8D9        FCOMP  ST(1)
0B9F EBEC        JMP    0B8D              ; Exit mit Status in AX

; ** Xref?  Vergleich TOS/R1 ohne POP
0BA1 9B          WAIT
0BA2 D8D1        FCOM   ST(1)
0BA4 EBE7        JMP    0B8D              ; Exit mit Status in AX
```

Ähnliches gilt für diese drei Subfunktionen, die jeweils TOS und R1 mit einer mehr oder minder großen Zahl anschließender POPs vergleichen. Da die Routine $BA1 nicht einmal in der Adresstabelle des $3E-Dispatchers eingetragen ist, ließe sich natürlich auch vermuten, daß wir es insgesamt mit dem (obsoleten) Rest einer nie vollständig realisierten Idee zu tun haben...

```
; ** INT $3E, Subcode $E8: FTST des TOS
0BA6 9B            WAIT
0BA7 D9E4          FTST
0BA9 EBE2          JMP    0B8D                  ; Exit mit Status in AX

; ** INT $3E, Subcode $EA: FXAM des TOS
0BAB 9B            WAIT
0BAC D9E5          FXAM
0BAE 9B            WAIT
0BAF DD7E06        FSTSW  [BP+06]               ; -> AX nach Rücksprung
0BB2 9B            WAIT
0BB3 C3            RET                          ; Exit direkt
```

Der 8086-Emulator bzw. eine Entschuldigung

Wenn man den von Turbo Pascal erzeugten .MAP-Dateien glauben darf, hat die im Modus {$E+} eingebundene Variante des 8087-Emulators rund den fünffachen Umfang seines 8087-Gegenstücks; mit dem entsprechenden Listing ließen sich also die nächsten 80 Seiten füllen - was ich weder mir noch dem Leser antun wollte*).

Ein paar Bytes zur Abschreckung:

```
****************************
* Dispatcher für $34..$3D *      8086-Version
****************************
2824 FB            STI
2825 FC            CLD
2826 52            PUSH   DX
2827 50            PUSH   AX
2828 53            PUSH   BX
2829 55            PUSH   BP
282A 56            PUSH   SI                    ; ALLE Register auf
282B 57            PUSH   DI                      den Stack des Programms
282C 06            PUSH   ES
282D 51            PUSH   CX
282E 1E            PUSH   DS
282F 8BEC          MOV    BP,SP
2831 36892EDE00    MOV    SS:[00DE],BP          ; inklusive SP
2836 C47612        LES    SI,[BP+12]            ; IRET-Adresse
2839 4E            DEC    SI
283A B200          MOV    DL,00
283C 26AD          LODSW  ES:                   ; INT-Nummer + 1. Byte
283E 2C34          SUB    AL,34                   (wie in der 8087-Version)
```

*) Dasselbe gilt für das Listing der »Shortcut«-Funktionen für *Sin*, *Cos* usw. - mathematisch Interessierte werden gebeten, die Startadressen-Tabelle des $3E-Dispatchers als Ansatzpunkt für eigene Forschungen zu benutzen.

```
2840 3C08        CMP    AL,08          ; >= INT $3C?
2842 73A8        JNB    27EC

2844 BB07C0      MOV    BX,C007
2847 22DC        AND    BL,AH
2849 22FC        AND    BH,AH          ; Ermittlung eines
284B 91          XCHG   AX,CX            Funktionscodes aus
284C 80FFC0      CMP    BH,C0            INT-Nummer + 1. Byte
284F 7367        JNB    28B8
2851 80FF40      CMP    BH,40
2854 7712        JA     2868
2856 7414        JZ     286C
2858 2BC0        SUB    AX,AX
285A 80FB06      CMP    BL,06
285D 7510        JNZ    286F
285F 26AD        LODSW  ES:            ; 2. und drittes Byte
2861 897612      MOV    [BP+12],SI     ; RET-Adresse erhöhen
2864 8CDE        MOV    SI,DS
2866 EB65        JMP    28CD
........
```

Diese Variante des Dispatchers ist relativ geradlinig programmiert*) und modifiziert das aufrufende Programm nur bei (hier natürlich unnötigen) WAIT-Befehlen. Ansonsten besteht dieses Modul aus einer fast unendlichen Folge mühsamster Bit-Analysen (INT-Befehl -> 8087-Opcode -> 8086-Sprungtabellen) und »zu Fuß«-Operationen (sprich: dem Normalisieren, Denormalisieren, Umformatieren, Vergleichen, Anpassen usw. von Exponenten und Mantissen) - also genau den Dingen, die in Kapitel 2 als »abgeschafft« bejubelt wurden.

*) Jedenfalls die Teile, die ich mir (flüchtig) angesehen habe.

Anhang D
Zusammenfassung, Nachgedanken & Tips

Dieser Anhang besteht aus drei Teilen:

- einer Zusammenfassung der wichtigsten Punkte der Kapitel 1 bis 6 mit einigen Erweiterungen und Ideen,
- einigen allgemeinen Ideen und/oder Erkenntnissen, die sich thematisch nicht oder nur schlecht anderswo unterbringen ließen,
- und einer ausführlichsten Beschreibung des einzigen Fehlers, der mir in Turbo Pascal aufgefallen ist.

Kapitel 1 - Speicherstrukturen

- Die Prüfung, ob einem Programm genügend Platz für den Heap zur Verfügung steht (*HeapMin*), erfolgt nicht durch die Laufzeitbibliothek, sondern durch DOS (Seite 5).

Der vorangehende Abschnitt (»Ein Programm im Speicher«) behauptet zwar, daß der transiente Teil von COMMAND.COM gegebenenfalls durch die Fragmentliste überschrieben wird, hüllt sich über das Wann aber in Schweigen: COMMAND.COM belegt die obersten rund 4 KByte des Hauptspeichers mit internen Datenpuffern, deren Inhalt vernünftigerweise nicht in die Prüfsummenbildung mit eingeht. Wie sich mit einem kleinen Programm (und genauer Beobachtung des Controller-Lämpchens) zeigen läßt, ist die unzusammenhängende Freigabe von rund 480 dynamischen Variablen erforderlich, damit *Programm*teile von COMMAND.COM überschrieben werden. Ergo: das »notwendige Nachladen von COMMAND.COM« ist auch bei Turbo Pascal-Programmen nicht der Regelfall, sondern eher die Ausnahme.

- Der Compiler belegt das Datensegment in der Reihenfolge:

```
typisierte Konstanten des Pascal-Programms
typisierte Konstanten von Units
Variablen des Pascal-Programms
Variablen von Units
```

  Da *Graph* (als einziges der 4.0-Standard-Units) eigene typisierte Konstanten deklariert, beginnen die Variablen des Pascal-Programms nach der Aufnahme dieses Units nicht mehr mit der Adresse 0000 im Datensegment (Seite 19). In der Version 5.0 definiert auch *System* eine Reihe typisierter Konstanten - hier muß der Programmierer also grundsätzlich mit Verschiebungen leben.

- *System* setzt unter anderem auch den Vektor $24 (»Critical Error«) auf eine eigene Behandlungsroutine um und macht damit spezielle Fehlerbehandlungsroutinen überflüssig (Seite 25).

Ein Detailunterschied der beiden Versionen, der sich hauptsächlich bei Programmierfehlern bemerkbar machen dürfte: In der Version 4.0 werden von der Laufzeitbibliothek belegte Interruptvektoren über Exit-Prozeduren zurückgesetzt, in der Version 5.0 dagegen durch eine direkte Schleife in der Prozedur *Halt*.

- Die Namen der Routinen von *System* erscheinen als einzige nicht in einer .MAP-Datei, obwohl die entsprechenden Routinen natürlich global verfügbar sind. Auf diese Weise haben die Programmierer von Borland .MAP-Dateien einigermaßen übersichtlich gehalten - und die Verfolgung von Programmen stark erschwert (Seite 27).
- Nur für die Version 5.0: Overlay-Routinen werden über den Interrupt $3F aufgerufen, der von *System* bis zur Initialisierung der dazugehörigen Verwaltung erst einmal auf eine Abbruchroutine gesetzt wird (Seite 31).
- Der Interrupt $10 mit dem Funktionscode $11 liefert offensichtlich bei EGA- und VGA-Karten die Zeilenzahl zurück (Seite 37).
- Das Unit *Printer* enthält keinen eigenen Gerätetreiber zur Ansteuerung der Drucker-Schnittstelle, sondern öffnet lediglich eine DOS-Datei mit dem Namen LPT1 (Seite 38 und Kapitel 6).
- Die Dateinamen CON, LPT1, LPT2 und AUX sind nach wie vor reserviert - im Gegensatz zu OUT, TRM und LST (Seite 39).
- Typisierte Konstanten werden ab der Version 4.0 im Datensegment gespeichert, untypisierte Konstanten dagegen im Codesegment (Seite 43).
- Als *far* deklarierte Routinen ruft der Compiler nur dann auch mit CALL FAR auf, wenn sie sich nicht innerhalb des »momentanen« Codesegments befinden. Ansonsten wird das CS-Register direkt auf den Stack gebracht und mit CALL (NEAR) gearbeitet (Seite 50).
- Funktionen reservieren lokal Speicherplatz für die Zwischenspeicherung ihres Ergebnisses - und zwar vor allen anderen lokalen Variablen (Seite 50).
- Für das Ergebnis einer Stringfunktion reserviert der Compiler grundsätzlich 256 Bytes »temporären« Speicherplatz - unabhängig davon, ob die Stringfunktion mit einem »kürzeren« Ergebnistyp arbeitet (Seite 56 ff).
- Strukturierte Wertparameter werden als Zeiger übergeben. Der Compiler stellt den entsprechenden Routinen einen speziellen Startcode voran, der lokal Speicherplatz belegt und eine »Lokalkopie« des übergebenen Wertes erzeugt (Seite 70).

- **inline** erzeugt auch für lokale Variablen und übergebene Parameter 16-Bit-Werte und bringt deshalb ein Programm zum Absturz, das mit »kurzen« Offsets (`MOV AX, [BP+02]` anstelle von `MOV AX,[BP+0002]`) zur Adressierung eines Parameters arbeitet (Seite 77).
- **const**-Deklarationen werden vom Compiler abhängig von ihrem Wert mit einem, zwei oder vier Bytes gespeichert und auch mit dieser Bytezahl in **inline**-Anweisungen eingesetzt. Wenn der entsprechende Prozessorbefehl eine andere Wortlänge erwartet, ergeben sich dieselben Probleme wie bei der Adressierung lokaler Variablen (Seite 83).
- Der Compiler ignoriert formale Parameternamen bei als **inline** deklarierten Routinen. Falls globale Variablen gleichen Namens deklariert sind, werden die entsprechenden Adressen ohne Rückmeldung eingesetzt (Seite 84).
- Routinen, die vollständig in Maschinensprache geschrieben sind, können strukturierte Wertparameter »getarnt« als **var** behandeln. Damit sollte man allerdings ausgesprochen vorsichtig sein (Seite 92).

Kapitel 2 - Der Coprozessor 8087

- Turbo Pascal macht (wie eigentlich alle Hochsprachen-Compiler) eine echte Parallelverarbeitung numerischer Werte durch den 8087 mit angefügten WAIT-Befehlen zunichte (Seite 97).

Was der entsprechende Abschnitt in Kapitel 2 unterschlägt: ab dem 80286/87 läuft die Synchronisation intern, explizit eingesetzte WAITs *vor* einem Coprozessor-Befehl sind also unnötig. Da der Compiler aber keine »Vorausanalyse« durchführt und deshalb nicht weiß, ob der 80286 direkt *nach* einem Coprozessor-Befehl auf dessen Ergebnisse zugreifen will, bleibt die Problematik dieselbe.

Ein WAIT-Befehl wird bereits auf einem 6 MHz-AT in weniger als 500 Nanosekunden ausgeführt, wenn es nichts zu warten gibt - deshalb haben sich die Programmierer von Borland auch bei Turbo C (mit seiner Option »286«) nicht gesondert um das Aussieben unnötiger WAIT-Befehle gekümmert.

- Die Funktion des Interruptvektors $75 ist in beiden Versionen ungeklärt (Seite 107).
- Der 8087-Emulator der Version 5.0 ist auch im Modus {$E-} grundsätzlich mit von der Partie, weil der Compiler in keinem Fall direkte 8087-Opcodes erzeugt (Seite 109ff und Anhang C).
- Der Compiler läßt Fließkomma-Wertparameter in der Version 4.0 zwar durch die Register des Coprozessors übergeben, erzeugt aber in derselben Weise wie bei Strings, Records und Arrays zusätzlichen Code für eine »Lokalkopie« zu Beginn der aufgerufenen Routine (Seite 119). In der Version 5.0 verwenden

Fließkomma-Wertparameter den 8086-Stack und machen entsprechende »Lokalkopien« überflüssig.

- Der Compiler behandelt den Datentyp *Real* nicht als »strukturiert«: Wert-Übergaben geschehen direkt (und ohne Zeiger), die Zuweisung untypisierter Konstanten wird mit direkten Prozessorbefehlen erledigt (Seite 133).
- *Real*-Variablen sind zusammen mit dem Schalter {$N+} eine ziemliche Katastrophe (Seite 138).
- Ein Herabsetzen der internen Rechengenauigkeit des 8087 bringt praktisch keinen Geschwindigkeitszuwachs (Seite 141).
- Die einzig vernünftige Alternative zu *Single* ist *Extended* - *Double*-Variablen werden am langsamsten verarbeitet (Seite 143).
- Die direkte Programmierung des 8087 bringt umso weniger, je komplexer der Rechenausdruck ist: Turbo Pascal lastet den Coprozessor so weit aus, wie es einer Hochsprache überhaupt möglich ist (Seite 157).

Kapitel 3 - Zeiger und Rekursion

Explizite Typ-Umwandlung von Zeigern

Die Typ-Umwandlung der »Quelle« einer Zuweisung (also der rechten Seite) beeinflußt normalerweise nur die Art und Weise, wie der Compiler das Ergebnis behandelt:

```
var x: Byte; y: Integer;
 ...
  y := x;                { liest ein Byte }
  y := ShortInt(x);      { liest ein Byte, erweitert das Vorzeichen auf
                           Integer und weist das Ergebnis y zu }
```

Bei Zeigervariablen erfolgt die Umwandlung dagegen in zwei Stufen: Der Typ des Zeigers bestimmt, wieviele Bytes der Prozessor liest (und läßt sich explizit umwandeln), mit einer zweiten Typ-Umwandlung kann man die Interpretation des Ergebnisses festlegen.

```
type   WordPtr: ^Word;
var
  y: Integer; z: Integer;
  BytePtr: ^Byte;     { Zeiger auf ein Byte }
  ...
  BytePtr := @z;      { auf die Adresse von z }
  y := BytePtr^;            { liest ein Byte }
  y := WordPtr(BytePtr)^;   { verwandelt BytePtr(!) in einen Zeiger auf
                              Word-Variablen, d.h. liest zwei Bytes }
  y := Integer(WordPtr(BytePtr)^);
```

Die letzte Zuweisung treibt die Sache sozusagen auf die Spitze: *BytePtr* wird in einen Zeiger auf *Word* konvertiert (der Prozessor liest also zwei Bytes); danach interpretiert das Programm die beiden gelesenen Bytes als *Integer* (und nicht als *Word*).

Obwohl eine Zeigervariable mit ihren beiden Feldern *Segment* und *Offset* eigentlich in die Klasse »strukturiert« fallen müßte, wird sie vom Compiler wie ein *LongInt* behandelt - Typ-Umwandlungen eines »kleineren« Ergebnisses setzen den »höherwertigen« Teil auf 0. Die Typ-Umwandlung einer Zeigervariablen, die ihrerseits auf einen Zeiger zeigen soll, fällt logisch konsequent, aber furchtbar kompliziert aus:

```
var
  XVec: Pointer; p: ^Word;

begin
  ....
  XVec := Pointer(p^);     { liest 2 Bytes und setzt XVec.Seg auf 0 (!) }
  XVec := Pointer(p)^;           { ergibt eine Fehlermeldung }
  XVec := Pointer(Pointer(p)^);  { liest vier Bytes }
  ...
```

Bei der zweiten Zuweisung wird *p* in einen Zeiger »auf gar nichts« umgewandelt - deshalb die Fehlermeldung.

Kapitel 4 - Speicherresidente Programme und Interrupts

(Hier wäre eigentlich als erstes eine boshafte Bemerkung über einen Artikel fällig, der in der Ausgabe 2/88 meiner Lieblingszeitschrift c't auf einem halben Dutzend Seiten erläutert, wie man speicherresidente Programme mit der Version 3.0 von Turbo Pascal schreibt und sie im Holzhammerverfahren über inoffizielle DOS-Funktionen wieder exorziert. Da die beiden Autoren aber an den Stack gedacht haben und sich Kollegen untereinander nicht ärgern sollen, entfällt die Bemerkung).

Die wichtigsten Punkte des Kapitels 4 in der Reihenfolge der Seitennummern:

- Programme, die mit einem Aufruf von KEEP enden, dürfen nicht von der integrierten Entwicklungsumgebung aus gestartet werden (Seite 231).
- *HeapMax* muß auf jeden Fall begrenzt werden - ansonsten belegt das Programm den gesamten Speicher und läßt nicht einmal mehr Platz für den transienten Teil von COMMAND.COM (Seite 232).
- Der Compiler sorgt bei **interrupt**-Routinen zwar für eine Zwischenspeicherung der Register und setzt auch das Datensegment des Pascal-Programms, »vergißt« aber den Stack. **interrupt**-Routinen sind deshalb nur für Männchen

auf dem Bildschirm brauchbar - oder für andere Aufgaben vergleichbarer Komplexität (Seite 236 f).

- Interrupt-gesteuerte Programme dürfen zwar keine DOS-Aufrufe ausführen - da die Ausgaberoutinen von *Crt* aber direkt in den Bildspeicher schreiben bzw. BIOS-Aufrufe benutzen, sind *Write* und *Writeln* nach der Aufnahme von *Crt* auch hier verwendbar (Seite 242).

- Ein TSR-Programm, das sich auch wieder aus dem Speicher entfernen läßt, muß:

  - den Originalwert des ersetzten Interruptvektors speichern;
  - die Startadresse seines Programmsegment-Präfix speichern;
  - eine Prüfung enthalten, ob bereits eine Version seiner selbst im Speicher installiert ist.

  Der vorgestellte Lösungsansatz geht davon aus, daß gespeicherte Daten in identischen Versionen zweier Programme dieselbe Relativposition zur Interrupt-Behandlungsroutine haben: Über den Wert des Interruptvektors lassen sich diese Daten also auch »von außen« ermitteln (Seite 252). Dieses Verfahren funktioniert allerdings nur dann, wenn der Vektor direkt auf das Programm zeigt - also nicht bei Interrupt-Ketten (Seite 260).

- Programme, die den Interruptvektor von DOS besetzen, können ihrerseits DOS-Aufrufe ausführen (Seite 262).

- BIOS-Funktionen lassen sich auch dann abfangen, wenn für ihren Ersatz DOS-Aufrufe notwendig sind (wie z.B. das Speichern von Druckerausgaben in einer Datei): die Daten werden (ohne DOS-Operationen) in einem Puffer zwischengespeichert und beim nächsten (abgefangenen) DOS-Aufruf geschrieben bzw. bearbeitet (Seite 270).

- Nur für die Version 5.0: Die vom Emulator belegten Interruptvektoren lassen sich auf einem System mit installiertem Coprozessor hinauswerfen, nachdem der entsprechende Programmteil mindestens einmal durchlaufen wurde - allerdings nur mit alleräußerster Vorsicht (Seite 287).

- Über direkte Zugriffe auf den Tastaturpuffer lassen sich Tastendrücke simulieren (Seite 306 ff).

Mehrere und simultane Interrupts mit einem TSR-Programm

Wie in Kapitel 4 erläutert, führt die Unterbrechung eines TSR-Programms »durch sich selbst« zum Systemabsturz, weil die zweite Unterbrechung wieder denselben Stack setzt (und damit lokale Variablen, Rücksprungadressen etc. des unterbrochenen Programms überschreibt).

Zwei Möglichkeiten wären zur Lösung dieses Problems denkbar:

- *statische Unterteilung des Stacks*: wenn das Programm beispielsweise drei Interruptvektoren besetzt, existiert für jeden dieser Vektoren ein eigenes »Rahmenprogramm« (INTFRAME, DOSHOOK oder DOSCATCH) zusammen mit einem eigenen Satz von Variablen für die Zwischenspeicherung des momentanen Stacks (*SSSave, SPSave*) - und einem eigenen Satz von Variablen für das Neusetzen von SS:SP (*TurboSS1, TurboSP1, TurboSS2, TurboSP2* usw). Diese Variablen werden zu Beginn des Pascal-Programms folgendermaßen initialisiert:

```
{$M 3000,0,0}    { 3000 Bytes Stack }
...
TurboSS1 := SSeg;   TurboSS2 := SSeg;    TurboSS3 := SSeg;
TurboSP1 := $1000;  TurboSP2 := $2000;   TurboSP3 := $3000;
   {      unteres            mittleres       oberes Drittel }
```

  Auf diese Weise hat jede der Interrupt-Behandlungsroutinen einen eigenen Stack-Bereich zur Verfügung.

- *dynamische Unterteilung des Stacks:* jede Interrupt-Behandlungsroutine hat die folgende Form:

```
procedure Int1Proc(var Regs:Registers);
var ....
begin
  Dec(TurboSP, $1000);    { für folgende Interrupts }
  inline($FB);        { STI - Interrupts an }
  ...
  < Rumpf der Prozedur >
  ...
  inline($FA);        { CLI - Interrupts aus }
  Inc(TurboSP, $1000);    { Stack wieder zurück }
end;
```

  Ein Interrupt, der während der Ausführung von *Int1Proc* stattfindet und innerhalb desselben TSR-Programms bearbeitet wird, verwendet also einen »darunterliegenden« Stack-Bereich.

  Achtung: Interrupts dürfen hier durch den »Rahmencode« nicht wieder freigegeben werden (d.h. das STI nach dem Umsetzen des Stacks entfällt). Ein »Turbo«-Interrupt, der *Int1Proc* vor der Erniedrigung von *TurboSS* bzw. nach der abschließenden Erhöhung dieser Variablen (vor dem Rücksprung) unterbricht und seinerseits *TurboSS* verändert, würde nämlich auch diese Systematik gründlich durcheinander bringen.

Kapitel 5 - Grafik

- *Graph* besteht aus mehreren Codesegmenten, die eine Art logischer Unterteilung des Programms repräsentieren (Seite 319ff).
- *Release* darf nicht mit *HeapOrg* als Parameter verwendet werden, wenn *Graph* aktiv ist - ansonsten ist es um den Grafiktreiber und/oder die Zeichensätze geschehen (Seite 323).
- Grafiktreiber und Zeichensätze enthalten eine »Pufferzone«. Sie werden von *InitGraph* bzw. *SetTextStyle* so im Codesegment umhergeschoben, daß sich die Startadresse des »eigentlichen Dateiinhalts« in der Form *Segment:0000* darstellen läßt. Der Funktions-Dispatcher eines Grafik-Treibers beginnt mit der Adresse 0000 dieses Segments, d.h. wird über eine eigene Segmentadresse angesprochen (Seite 328).
- *InitGraph* »patcht« einen geladenen Grafiktreiber mit der Adresse einer Routine von *Graph* (Seite 331).
- Initialisierungen der Grafik-Karte geschehen ausschließlich im geladenen Grafik-Treiber, der auch die Eckwerte des momentan gesetzten Bildformats festlegt. Eine Anpassung für Grafik-Karten, die 640 * 480 Bildpunkte unterstützen, aber eine eigene Initialisierung erwarten, ist problemlos (Seite 334f).
- Grafik-Zeichensätze sind schrecklich kompliziert aufgebaut (Seite 343 ff), wegen einer in *OutText* durchgeführten AND-Operation lassen sie sich nur in Richtung auf niedrigere ASCII-Codes erweitern (Seite 353 ff).

Kapitel 6 - Dateivariablen und Gerätetreiber

- Dateivariablen sind ab der Version 4.0 so etwas wie eine universelle Zwischenstation für Daten jeder Art, weil sie neben einer Zustandsbeschreibung der Datei auch die Startadressen der Routinen enthalten, über die die Kommunikation abgewickelt wird (Seite 410).
- Drucker-Ausgaben lassen sich mit einem eigenen Gerätetreiber gegenüber dem Unit *Printer* auf mehr als das Dreifache beschleunigen*) (Seite 412).
- Die Treiber-Routinen von *Input* und *Output* lassen sich auf dieselbe Weise ersetzen wie bei anderen Dateivariablen (Seite 416).
- *Write* und *Writeln* arbeiten im Normalfall nicht über *InOut*, sondern über *Flush* (Seite 417).

*) Was vor lauter Begeisterung über das Unit *FastList* völlig unter den Tisch gefallen ist: Ein Aufruf der BIOS-Funktion anstelle des direkten Portzugriffs innerhalb der *InOut*-Funktion bringt immer noch einen Geschwindigkeitszuwachs von rund 100 Prozent.

- Das Vergrößern eines Textpuffers bringt bei Ausgaben keine Steigerung der Geschwindigkeit (Seite 417).
- Ein »Dazwischenschalten« eigener Routinen bei Ein- und Ausgaben ist ähnlich wie bei Interrupts über Vektorketten möglich (Seite 418 f).
- Die Variablen *ConInPtr* und *ConOutPtr* der Version 3.0 lassen sich zu hundert Prozent in einem 4.0- bzw. 5.0-Programm nachbilden (Seite 420).

Allgemeines

Formatierte Ausgabe von Nullstrings

Die Laufzeitbibliothek von Turbo Pascal behandelt bei der formatierten Ausgabe über *Writeln* auch Nullstrings auf korrekte Weise:

```
Writeln(' ':20);     { ergibt 20 Leerzeichen }
Writeln('':20);      { dito }
```

Im zweiten Fall wird ebenfalls eine untypisierte Konstante im Codesegment gespeichert - nämlich ein Byte mit dem Wert 0.

Start- und Endwerte von for-Schleifen...

werden von Turbo Pascal nur ein einziges Mal berechnet:

```
var x,b: Integer;
begin
  b := 10;
  for x:= 1 to b do
  begin
    b := 5;  Write(x:2);     { ergibt 1 2 3 4 5 6 7 8 9 10 }
  end;
end.
```

Abgesehen davon, daß eine **for**-Schleife also wirklich nur dann verwendet werden kann, wenn die Anzahl der Durchläufe vorher feststeht, ergibt sich eine erfreuliche Folgerung: Weil der Compiler die Eckwerte in jedem Fall nur einmal berechnen läßt, steht nicht zu befürchten, daß ein Ausdruck wie

```
for x:= 1 to Length(StrVar1+StrVar6) do  ....
```

bei jedem Schleifendurchlauf erneut Rechenarbeit verursacht.

MaxColors...

ist (im Gegensatz zu *MaxX* und *MaxY*) eine Konstante. Der Compiler setzt hier einfach den Wert 16 ein und überläßt den Routinen des Grafiktreibers die eventuell notwendige Begrenzung.

Compilerfehler

Für die Version 4.0 bzw. 5.0 eines Programms gehört sich eine gewisse Freiheit von Fehlern - tatsächlich ist mir in mehr als einem Jahr intensiver Arbeit mit Turbo Pascal nur der Linker als Quelle möglicher Probleme aufgefallen*):

Sprungadressen im Datensegment

Zuerst das Maschinenprogramm - diesmal in der Form, wie es MASM als Listing liefert:

```
 1 0000                 DATA   Segment Public Word
 2                             EXTRN CallAddrs: DWord      ; Pascal-Variable
 3 0000                 DATA   ends
 4 0000                 CODE   Segment Public Byte
 5                             ASSUME cs:CODE, ds:DATA
 6                             Public CallP
 7 0000                 CallP  proc near
 8 0000  FF 1E FFFE E          CALL FAR [CallAddrs]
 9 0004  C3                    ret
10                      CallP  endp
11 0005                 CODE   ends
12                             end
```

Die Routine *CallP* sollte also eine Routine aufrufen, deren Startadresse in der Pascal-Variable *CallAddrs* gespeichert ist. Da MASM die Adresse dieser (externen) Pascal-Variablen nicht kennt, wird der CALL-Befehl entsprechend gekennzeichnet.

Ein Pascal-Programm dazu:

```
program CALLError;
var Dummy: array[1..100] of Byte;
    CallAddrs: Pointer;

procedure CallP; external;
{$L CALLERR}

{$F+} procedure GetsCalled; {$F-}
begin
  Writeln('Prozedur wurde aufgerufen!');
end;

begin
  CallAddrs := @GetsCalled;     CallP;
end.
```

*) Angesichts der starken Erweiterungen und der radikalen Änderung des Konzepts gegenüber der Version 3.0 kann man nur sagen: eine fantastische Leistung der Programmierer von Borland.

Und schließlich ein Blick mit SYMDEB auf das fertige Programm:

```
** begin
CALLERROR1:@:
003E 9A00006A3A     CALL   SYSTEM:@
0043 89E5           MOV    BP,SP
** CallAddrs := @GetsCalled;
0045 B81B00         MOV    AX,001B        ; Adresse von GetsCalled
0048 8CCA           MOV    DX,CS
004A A36400         MOV    [0064],AX      ; -> Variable CallAddrs
004D 89166600       MOV    [0066],DX
** CallP;
0051 E80900         CALL   CALLP
** end.
0054 89EC           MOV    SP,BP
   ....

CALLERROR:CALLP:
005D FF1E6200       CALL   FAR [0062]     ; *** sollte [0064] sein!
0061 C3             RET
```

Hier hat sich der Linker verzählt - was prinzipbedingt zu sein scheint: In der **Version 5.0** hat *CallAddrs* zwar eine andere Adresse innerhalb des Pascal-Programms ($00A0), das Problem ist aber dasselbe geblieben: der Aufruf durch das Maschinenprogramm liest in dieser Version die Speicherzellen $009E..A1 des Datensegments.

Adressen von Pascal-Routinen im Codesegment

Wenn man den Linker dazu zwingt, Adressen von Pascal-Prozeduren im Codesegment eines Maschinenprogramms abzulegen, reagiert er ebenfalls unwillig. Auch hier zuerst ein von MASM erzeugtes Listing:

```
 1 0000                    CODE   Segment Public Byte
 2                                ASSUME cs:CODE
 3                                EXTRN  GetsCalled: far   ; Pascal-Routine
 4                                Public CallP
 5 0000  0000 ---- E       CallX  dd GetsCalled
 6 0004                    CallP  proc near
 7 0004  2E: FF 1E FFFE R         CALL FAR [CallX]
 8 0009  C3                       ret
 9                         CallP  endp
10 000A                    CODE   ends
11                                end
```

Die Adresse der Pascal-Routine *GetsCalled* sollte hier also ab *CallX* eingesetzt werden. Das Pascal-Programm dazu ist fast dasselbe wie im vorigen Beispiel:

```
program CALLCSError;
var
  Dummy: array[1..100] of Byte;

procedure CallP; external;
{$L CallCS.OBJ}

{$F+} procedure GetsCalled; {$F-}
begin
  Writeln('Prozedur wurde aufgerufen!');
end;

begin
  CallP;
end.
```

Und das Ergebnis:

```
CALLCSERROR1:@:
** begin
3A62:003E 9A0000683A     CALL   SYSTEM:@
3A62:0043 89E5           MOV    BP,SP
**  CallP;
3A62:0045 E80D00         CALL   CALLP
15:     end.
       ....

CALLCSERROR:CALLP:
3A62:0055 2EFF1E4F00     CALL   FAR CS:[004F]
3A62:005A C3             RET

-dw cs:4f
3A62:004F  3A68 001B
```

Wie sich zeigen läßt, steht *GetsCalled* tatsächlich ab der Adresse $001B im Codesegment. Der Offset-Anteil des vom Linker eingesetzten Wertes ist also korrekt - im Gegensatz zum Segment, das hier (aus welchen Gründen auch immer) um sechs »Paragraphs« zu hoch angesetzt ist.

Die **Version 5.0** ist leider auch hier hundertprozentig »kompatibel« - sie verwendet zwar andere Adressen, produziert aber genau denselben Unsinn.

Ein letzter **Hinweis:** Die Bezugnahme auf Startadressen, die sich innerhalb des Assembler-Programms befinden, z.B.

```
CODE    Segment Public Byte
        ....
Table   dw Proc2, Proc3, Proc4  ; Startadressen der Routinen
Proc1   proc near
        mov bx,FuncNo
        shl bx,1
        jmp [Table+bx]
        ....
Proc2   proc near
        ...
```

funktioniert dagegen einwandfrei (genauso wie direkte Zugriffe auf Pascal-Variablen im Stile von `MOV AX,PasInt`).

Index

8086 (Prozessor) 425
8087 (Coprozessor) 93

A
Abbruchkriterien für Rekursion 193
Absuche von Directories
  nach Dateinamen 199
  über Datum/Uhrzeit 205
Adresse 426
  des ersten Programmbefehls 11
  Offset-Anteil 437
  Zerlegung in Segment und Offset 177
Adressierung
  Abhängigkeiten 79
  Daten vs. Adressen 451
  der 8087-Register 99
  Grafiktreiber in DEBUG 336
  lokaler Variablen 64
  mehrdeutige 437
  Partial- 83
  sichere über BP 83
  Variablenparameter 66
  von Bildpunkten 341
  von Variablen 164
  Wertparameter 63
Allzweck-Register 427
Anhängen von Zeichendefinitionen 354
Apfelmännchen 149
  mathematische Optimierung 149
  Version 5.0 155, 157
  Zeitbedarf 150
Arrays
  mit Zeigerelementen 181
  und Typ-Umwandlungen 173
ASCIIZ-Strings 292
Assign... 411
AssignCrt 33
  und Vektorketten 420
AssignKbd 42
ASSUME-Anweisung von MASM 448
Aufruf von MASM 443
Ausführung eines Programms 229
Ausgabe
  an den Drucker 412
  formatierte Nullstrings 475
  Umleitungsmöglichkeiten 416
  über Output 45
    TURBO 3.0 47
  Vektorketten 419
    mit Crt 420
  von Grafikzeichen mit Pascal 348
  zeichenweise 421
Ausnahmezustände für Fließkomma-Operationen 106
  Maskierung 104
    durch Iterate zurückgesetzt 155
  Version 5.0 107
AUX: (Spezialdatei) 39

B
Basisadresse des Bildspeichers, Berechnung 340
Beispielprogramme
  Apfel1 150
  Apfel3 157
  Catch22 291
  CE 356
  ConInOutPtr 421
  DateFind 205
  DrawChar 348
  EGAFile (Unit) 404
  ENVP.PAS 6
  FastList (Unit) 413
  FFClone 201
  HookDemo 267
  InOutLinkDemo 419
  IntFrameTest 241
  List1Demo 216
  List2Demo 224
  NewIODemo 417
  OUTTEXT.ASM 401
  Precision8087 142
  PrinterToFile 271
  ProcArray 187
  ProcFuncPtr 186
  RemoveableFunctionInterrupt 246
  Speed87 146
  TPCED 294
  TRACE287 279

TRACE86 242
TRACE86R 253
UpCase.ASM 86
Upper.ASM 89
Benchmarks für den 8087 146
Bildformate
beliebige 341
eigene 334
Parametertabellen in EGAVGA.BGI 338, 339
Standard- 334
von Grafiktreibern 333
Bildpunkte, Adressberechnung 341
Bildschirm
direkter Zugriff 169
Zeilenzahl 37
BIOS
Clock Tick 235
EGA vs. CGA 37
Funktionen abfangen 270
im System 2
Zeitmessung über »Ticks« 143
BP-Register 52, 65
relative Adressierung 79
sichere Adressierung 83
Vorsicht mit 53

C

CE (Zeichensatz-Editor) 356
Editor-Befehle 359
Hauptmenü 358
Hauptprogramm CE.PAS 386
Menü Zeichnen 358
Menüpunkte 358
Programmstruktur 361
Start 357
Unit CEEdit 375
Unit CEUtils 363
Unit GEGlobal 361
Zusatzfunktion ImportChar 396
CheckBreak, CheckEof 35
CheckSnow 33
Prüfung 37
»Clock Tick«-Interrupt 235
CloseFunc (Zeiger) 411
Close-Funktion, Beispiel 414
CmdLine (PSP-Eintrag) 7
Codesegment
des Pascal-Hauptprogramms 14
eines Grafiktreibers 320
eines TSR-Programms 252
für Maschinenprogramme 444
Grafiktreiber im 325
Pascal-Adressen 477
untypisierte Konstanten 44
COMMAND.COM 3
Absturz bei Interrupts 236
Lage im Speicher 3
Nachladen 467
und TSR-Programme 232
Compilerfehler 476
Compilierung, bedingte, mit Problemchen 280
ConInPtr und ConOutPtr, Ersatz 420
CON: (Spezialdatei) 39
const-Deklarationen
typisierte 18
Vorsicht mit inline 83
Control-Break
Austausch durch Crt 36
direkter DOS-Aufruf 308
und TSR-Programme 306
Vektor 26
Coprozessor 93
automatische Synchronisation 469
Benchmarks 146
direkte Programmierung 148
Emulator (Version 5.0)
Initialisierung 111
im Detail 453
Initialisierung 105
INT-Umwandlung 454
Konstanten F??RQQ 454
Kontrollwort 104
Prüfung auf »Vorhanden« 106
Rechengenauigkeit 141
Registerinhalte verfolgen 279
Stack 98
Statuswort 102
Tag-Register 100
»Critical Error«, Vektor 26
in TSR-Programmen 273
Crt (Unit)
Initialisierung 33
Funktion ReadKey 306

und Vektorketten 420
Writeln ohne DOS-Aufruf 242
CS-Register 428

D

Dateistrukturen
.COM 16
.EXE 10
Grafiktreiber 327
Zeichensätze 343
Dateivariablen
Begriffsdefinition 409
Entwicklungsgeschichte 409
Input, Output 419
LPT1 412
Struktur 410
Treiber-Routinen 411
TURBO 3.0 409
Datensegment
DS-Register setzen 23
eines TSR-Programms 252
in Maschinenprogrammen 447
Lage im Programm 11
TURBO 3.0 17
Delay, Initialisierung 35
denormalisierte Werte 101
in der Version 5.0 108
deutsche Umlaute in Zeichensätzen, Umcodierung 400
Directorystrukturen 195
Absuche über Datum/Uhrzeit 205
Einträge '.' und '..' 198
DirectVideo 33
direkte Programmierung des 8087 148
Dispatcher
eines Grafiktreibers 330
im Unit Graph 317
Dispose (Standardprozedur) 184
Doppelpunkte in Dateinamen 39
doppelt verkettete Listen 221
Beispielprogramm List2Demo 224
Einfügen und Löschen 222
DOS
Aufruf
durch Control-Break 308
in TSR-Programmen 262
nicht durch Crt.Writeln 242
wann nicht 262
Einhängen in 262
Funktionen
Ersatz 288
»Buffered Keyboard Input« 270
»Free Allocated Memory« 250
»Remove Directory« 291
KEEP 230
»nächste freie Adresse« 230
Speicherverwaltung 260
Dos (Unit), Initialisierung 33
Double (Datentyp), langsamer als Extended 143
Drucker
eigener Gerätetreiber 412
Umschaltung via Control-p 306
DS-Register 428
gesetzt durch System 23
Dummy-Element für Listen 223
dynamische Variablen 183
Begriffsdefinition 185
»freigegebene« 184
Grafiktreiber 321

E

Eckdaten
von TSR-Programmen 245
Ermittlung 252
von Zeigern 167
Konvertierung 176
Editor für Zeichensätze 355
EGAVGA.BGI
Funktion $00 337
Funktion $02 338
Funktion $30 341
Parametertabellen 338, 339
eigene
Bildformate für Grafiktreiber 334
Initialisierungen von Grafik-Karten 340
einfache Listen 213
Beispielprogramm List1Demo 216
Einfügen und Löschen 214
Eingabe
Umleitungsmöglichkeiten 416
zeichenweise 421
Emulator (Version 5.0) 453
Initialisierung 107
INT $3E 459
INT-Umwandlung 454

entfernbare TSR-Programme 245
EnvCount (Pascal-Funktion) 6
Environment-Tabelle 4
  in TSR-Programmen 250
Environ (PSP-Eintrag) 6
EnvStr (Pascal-Funktion) 6
Erdbeermarmelade 285
Evolution von Dateivariablen 409
.EXE-Datei, Kopfinformation 10
  vs. .COM-Dateien 16
ExitProc, Initialisierung 24
Exit-Prozedur von Crt 36
explizite Typ-Umwandlungen 172
Extended (Datentyp) vs. Double 143
EXTRN-Deklarationen 446

F

far-Aufrufe und Sprünge 438
  für Pascal-Routinen 50
  im selben Segment 50
  Relozierung 13
Fehlerbehandlung
  »Critical Error» 26
    in TSR-Programmen 273
  für den Heap 24
    durch Graph 40
    durch Graph, Version 5.0 41
  in 8087-Maschinenprogrammen 153
FILEFIND 195
FINIT (8087-Befehl) 105
FISRQQ..FIARQQ (Emulator-Konstanten) 454
FLD (8087-Befehl) 98
Fließkomma-Fehler in Maschinenprogrammen 153
Fließkomma-Konstanten
  untypisierte 45
    im Modus {$N+} 121
    TURBO 3.0 47
Fließkommawerte
  als Funktionsergebnisse 127
  als Parameter 119
    Version 5.0 125
  Rechengeschwindigkeit 141
  Vergleiche 103
FlushFunc (Zeiger) 411
Flush-Funktion
  bei Write 417
  Beispiel 419
FMULP (8087-Befehl) 98
for-Schleifen, Start- und Endwerte 475
forward-Deklarationen in Maschinenprogrammen 451
Fragmentliste
  Initialisierung 24
  und COMMAND.COM 467
FreeMem (Standardprozedur) 183
FreeMin, Erhöhung durch Graph 41
Freigabe
  eines dynamisch belegten Bereichs 184
  eines TSR-Programms 250
  von Interrupts 442
FRSTOR (8087-Befehl) 284
FSAVE (8087-Befehl) 282
FSTP (8087-Befehl) 99
Funktionen
  allg. Vergleich mit TURBO 3.0 53
  Ergebnis 50
    Fließkomma 127
  in Maschinensprache 86
  parameterlose 50
  Real- 132
    TURBO 3.0 135
  String- 56
    TURBO 3.0 60
    und Rekursion 194
  Unterschiede zu Prozeduren 74
    TURBO 3.0 76
  Zeiger auf 185
Funktionscodes für Zeichensatz-Vektorpaare 346
Funktionsnummern
  für Pascal-Routinen 187
  in Grafiktreibern 320, 330
  in TSR-Programmen 246
generische Zeigertypen 168

G

GENOA HIRES 334
Gerätetreiber (Begriffsdefinition) 409
Geschwindigkeit und Präzision 141
GetMem (Standardprozedur) 183
  und Strings 212
Grafik 315

Grafikbilder, Laden und Speichern 404
Grafiktreiber 327
  Berechnung von Bildpunkt-
    Adressen 341
  Dispatcher 330
  dynamisch geladen 321
  Lage im Speicher 320
  Startadressen-Tabelle 331
  statisch eingebunden 324
  und DEBUG 336
  Vektor 320
Grafik-Zeichensätze 343
  Ausgabe 348
  Bearbeiten 356
  Mischen von Dateien 396
Graph3 (Unit), Initialisierung 42
Graph (Unit) 315
  Initialisierung 39, 326
  mehrere Segmente 319
  Prüfung auf »Initialisiert« 319
  Struktur 315
  typisierte Konstantendeklaration 20
  und SYMDEB 321
  Vorsicht mit Release 323

H
Hardware-Interrupts 251
Hauptprogramm
  lokale Reservierung 57
  Name 15
  Rahmencode in V 5.0 11
Hauptspeicher, Organisation 1
  Segmentierung 434
  Unterteilung durch Programme 436
Heap
  Belegung 183
  Fehlerbehandlung durch Graph 40
    Version 5.0 41
  Lage im Speicher 15
    TURBO 3.0 17
  Setzen durch System 23
    Version 5.0 29
HeapError, Initialisierung 24
HeapMax
  im Kopf einer .EXE-Datei 10
  TURBO 3.0 17
  und HiMem 5
  und TSR-Programme 232
HeapMin
  im Kopf einer .EXE-Datei 10
  Prüfung 5
HeapOrg
  gesetzt durch System 23
    Version 5.0 29
  unverändert bei Graph 323
HiMem (PSP-Eintrag) 5

I
IBMBIO.COM 2
IBMDOS.COM 2
Inf 101
INIT87 (Coprozessor-Init) 106
  Version 5.0 112
InitGraph 326
  Umkopieren von Grafiktreibern 329
Initialisierung
  Crt und Dos 33
  der Fragmentliste 24
  der Units 21
  des 8087 105
  des Heaps 23
    Version 5.0 29
  des Systems 2
  des Videomodus 34
  eigene für Grafik-Karten 340
  für Delay 35
  Graph 326
  Overlay (Version 5.0) 31
  Printer 38
  System 22
    Version 5.0 28
  Textfenster 38
  Turbo3, Graph3 42
  von Grafik-Karten 336, 337
inkompatible Zeiger 168
inline 77
  Deklarationen 83
  für Ein-/Ausgabeketten 419
  für Funktionszeiger 186
  mit symbolischen Namen 80
  Vorsicht mit 77
InOutFunc (Zeiger) 411
InOut-Funktion
  bei Write 417
  Beispiel 419

Input (Dateivariable)
  Öffnen 26
  Umleitungsmöglichkeiten 416
  und AssignCrt 33
Interrupt 439
  Controller 441
  Sperren und Freigabe 442
    beim Umsetzen des Stacks 239
interrupt-Deklaration 233
  Vorsicht mit 236
Interrupt-Ketten 261
  Reihenfolge 262
Interruptvektoren 441
  $1B 308
    gesetzt von Crt 36
    Version 5.0 38
  $3E im Emulator (V 5.0) 459
  $34..3D im Emulator (V 5.0) 456
  8087-Emulator (V 5.0) 453
    Zurücksetzen (V 5.0) 287
  Adressen 2
  »Clock Tick« 235
    TRACE86 242
    TRACE86R 253
  Control-Break 26, 36
  »Critical Error« 26
    in TSR-Programmen 273
  DOS-Ersatz 288
  DOS-Vektor umleiten 262
  Einhängen zusätzlicher Funktionen 235
  Ersetzen 233
  für den 8087 107
  für Overlays 30
  mehrere für ein Programm 472
  mehrere Programme 261
  »Print Screen« 233
  Setzen von System 25
  Speichern in System, Version 5.0 30
  Speicherung des Originalwertes 245
  TURBO-eigene, Vorsicht mit 259
IOResult
  als Ergebnis einer Treiber-Routine 411
  Variable 24
    Version 5.0 28
IRET (8086-Befehl) 235

K
Kbd (Spezialdatei) 42
KEEP (DOS-Funktion) 230
  Parameter 231
  Vorsicht mit TURBO.EXE 231
Kennbytes eines TSR-Programms 249
Kommandozeilen-Editor 294
Kommandozeilen-Parameter 7
Konstanten 43
  F??RQQ 454
Kontrollwort-Register des 8087 104
Konvertierung Real -> Extended 139
Kopfinformation
  einer .EXE-Datei 10
  von Grafiktreibern 327
  von Zeichensätzen 343
Kreis-/Zykloidentest 149
  als Maschinenprogramm 157

L
Laden und Speichern von Grafikbildern 404
Laufzeitbibliothek von TURBO 3.0 16
Length optimiert 79
lineare Listen 211
Linker
  Merkwürdigkeiten mit Graph 327
  Pascal-Adressen im Codesegment 477
  Relozierung indirekter Sprünge 476
Listen
  doppelt verkettete 221
  Dummy-Element 223
  einfache 213
  leere 216
Listings
  8087-Emulator (V 5.0) 456
  APFEL1.PAS 151
  APFEL3.PAS 157
  CATCH22.PAS 291
  CEEdit (CE-Unit) 375
  CEGlobal (CE-Unit) 361
  CE.PAS 386
  CEUtils (CE-Unit) 363
  CONIOPTR.PAS 421
  CRT 33
  CYCLTEST.ASM 160
  DATEFIND.PAS 206
  DOSCATCH.ASM 288

DOSHOOK.ASM 264
DRAWCHAR.PAS 348
DSPENTRY.INC 204
EGAFILE.PAS 404
EGAVGA.BGI 331
FASTLIST.PAS 413
FFCLONE.PAS 201
Grafik-Initialisierung 337
GRAPH 40
HOOKDEMO.PAS 268
INIT87 in System 106
INIT87 in System, Version 5.0 112
INT2CHK.PAS 276
INTCHECK.INC 255, 256
INTFRAME.ASM 237
INTFTEST.PAS 241
IOLINK.PAS 419
ITER8087.ASM 153
LIST1DEMO.PAS 216
LIST2DEMO.PAS 224
NEWIOPAS 417
OUTEXT.ASM 401
OutText 353
PREC8087.PAS 142
PRINTER 38
PRTFILE.PAS 271
PutPixel im Grafiktreiber 341
REMFNINT.PAS 247
SPEED87.PAS 146
Startcode eines 3.0-Programms 17
SYSTEM 23
TPCED0.PAS 296
TPCED erweitert 309
TPCED.PAS 302
TRACE287.PAS 280
TRACE86.PAS 242
TRACE86R.PAS 253
TURBO3 42
UPCASE.ASM 86
UPPER.ASM 89
Lokales
  Abhängigkeiten der Adressierung 79
  Funktionsergebnisse 53, 133
    TURBO 3.0 60
  Kopien von Wertparametern 70
  Real-Variablen 132
  Reihenfolge der Reservierung 77
    im Modus {$N+} 127
  Reservierung im Hauptprogramm 57
  Variablen 64
    Reihenfolge 66
    TURBO 3.0 66, 69
Lokalkopien
  und Rekursion 194
  von Fließkommawerten 120
  von Strings 70
Löschen von Zeichendefinitionen 354
LPT1 (DOS-Geräteeinheit) 39, 412
Lst (Dateivariable) 39
LST: (Spezialdatei) 39

M
Mark (Standardprozedur) 184
Maschinenprogramme
  CODE-Segment 444
  DATA-Segment 447
  Daten vs. Adressen 451
  forwards 451
  für Apfelmännchen 153
  Prozeduren und Funktionen 86
  Variablengrößen 450
MASM 443
  Datendeklarationen 447
  Datengrößen 449
  Deklarationen 446
  Programmcode 444
  Registers-Deklaration 237
mathematische Optimierung für
  Apfelmännchen 149
MaxColors 475
mehrfache Installation von TSR-
  Programmen 245
  bei Vektorketten 262
Mischen von Grafik-Zeichensätzen 396
Modus {$E} (Version 5.0) 109ff
Modus {$N-}, mit Real-Variablen 132
Modus {$N+}
  Initialisierung 105
  Interruptvektoren 107
  Real-Variablen 138

N
Namen
  bei inline-Deklarationen ignoriert 84
  der Unit-Segmente 21
  Hauptprogramm 15
  in System 27, 468

mit inline 80
Segmente
Crt und Dos 33
Turbo3 und Graph3 42
NaN (Not a Number) 102
near-Aufrufe und Sprünge 438
für Pascal-Routinen 50
mit MASM 446
Relozierung 12
New (Standardprozedur) 183
Normalisierung von Zeigern 438
Notation von Zeigervariablen 168
Nullstrings
Namen für Input, Output 26
Nullstrings, formatierte Ausgabe 475

O

Offset 437
für Sprünge 12
Offset in einer Adresse 177
Ofs (Standardfunktion) 178
OpenFunc (Zeiger) 411
Open-Funktion
Anforderungen 412
Beispiel 414
Optimierung
für 8087-Programme 148
von Zeichendefinitionen 355
Organisation des Hauptspeichers 1
Output (Dateivariable)
Öffnen 27
Umleitungsmöglichkeiten 416
und AssignCrt 34
OutText (Standardprozedur)
AND-Operation 352
Ersatz 353, 400
OverHeapPtr (Version 5.0), im
Initialisierungsteil von System 29
Overlays (Version 5.0)
Prinzip 31
Puffer festlegen 29
Vektor $3F, Initialisierung 30
Vektor in TSR-Programmen 232
Overlay (Unit), Initialisierung 31
OvrHeapOrg (Version 5.0), im
Initialisierungsteil von System 29
OvrHeapSize (Version 5.0), im
Initialisierungsteil von System 29

P

Parallelverarbeitung 95
Parameter 62
einfache Werte 63
gemischte 123
keine Namen bein inline-
Deklarationen 84
Real- 132
TURBO 3.0 135
Single..Extended 119
Version 5.0 125
strukturierte Werte 70
TURBO 3.0 66
Umwandlung untypisierter 173
und inline 81
var 66
im Modus {$N+} 130
und Rekursion 193
Parametertabellen für
Bildformate 338
Grafiktreiber 333
Partialadressierung 83
Pointer (Variablentyp) 168
Umwandlung 172
PrefixSeg, gesetzt durch System 23
Printer (Unit)
Ersatz 412
Initialisierung 38
»Print Screen«-Interrupt 233
in TRACE287 281
Programm 426
Adresse des ersten Befehls 11
Ausführung 4, 229
im Modus {$N+} 98
Version 5.0 108ff
Lage des Datensegments 11
Lage im Hauptspeicher 4
legale Selbstunterbrechung 472
Reihenfolge der Units 18
Segmente 8
selbstunterbrechend 244
legal 472
Startwerte der Register 9
Struktur 8, 13
bei der Aufnahme von Graph 321
TURBO 3.0 16
Version 5.0 15
typisches 13

Unterteilung des Hauptspeichers 436
Variablen und Konstanten 18
Programmsegment-Präfix (s. PSP) 4
Prozeduren 50
allg. Vergleich mit TURBO 3.0 50, 66
in Maschinensprache 86
mit Fließkommawerten 119
parameterlose 50
Unterschiede zu Funktionen 74
TURBO 3.0 76
Zeiger auf 185
Prozessor 8086 425
Programmiermodell 425
Register 425
Pseudo-var-Parameter 92
PSP 4
Ermittlung bei TSR-Programmen 250
Struktur 5
TURBO 3.0 16
Ptr (Standardfunktion) 173
PWord (selbstdefinierter Zeigerzerlegungsrecordtyp) 177
vs. Seg und Ofs 178

R

RandSeed, Initialisierung 24
ReadKey (Standardfunktion), neue Version 306
Real (Datentyp)
direkte Speicherung 132
im Modus {$N+} 138
Konvertierung nach Extended 139
Zeitvergleich mit Extended 147
Rechenregister des 8087 99
Records
für MASM 237
zur Typ-Umwandlung 176
Beispiel 176
Register 426
des 8086, 8087 97, 8086 425
Setzen beim Programmstart 9
Registrierung von Grafiktreibern 324
Reihenfolge lokaler Belegungen 77
Rekursion 191
hierarchische Directories 195
»Untereinanderstapeln« 193
kombiniert mit Schleifen 199
und var-Parameter 193
Release (Standardprozedur) 184
Vorsicht mit Graph 323
Relozierung 12
von Grafiktreibern 329
Vorsicht mit indirekten Sprüngen 476
RESET-Ablauf 2
residenter Teil von COMMAND.COM 3
RET $xxxx 55, 65
Rücksprungadressen
direkt vom Stack 87
nicht bei inline-Deklarationen 84

S

SANS.CHR (Zeichensatz), Besonderheiten 354
Segment-Anteil 437
Segmente
Anteil in einer Zeigervariablen 177
eines Programms 8
eines TSR-Programms 252
gemeinsame für Units 21
gemeinsames für Turbo3 und Graph3 42
mehrere in Graph 319
von Overlay-Units 31
Segmentierung des Speichers 434
Segmentregister 428
Seg (Standardfunktion) 178
selbstunterbrechende Programme 244
legal 472
SetGraphMode (Standardprozedur) 335
SetIntVec (Standardprozedur) 233
SetTextBufSize (Standardprozedur) 418
Shortcut-Interrupt 459
Single (Datentyp)
implizite Konvertierung (V 5.0) 125
Rechenbereich 101
vs. Extended 141
Skalierung von Zeichensätzen 347
Sperren von Interrupts 442
beim Umsetzen des Stacks 239
Spezialdateien
CON:, AUX:, LST: 39
Kbd 42
LPT1 39, 412
Spezialregister 428

Stack 429
beim Start des Programms 9
BP-Register 65
des 8087 98
durch Interrupts nicht gesetzt 236
dynamische Unterteilung für
Interrupts 473
Funktionsergebnisse 50
TURBO 3.0 53, 35
Größe
im Kopf einer .EXE-Datei 11
Umrechnung für HeapOrg 23
Version 5.0 29
im Modus {$N+} 125
lokale Variablen 64
Parameter 64
Real-Variablen
TURBO 3.0 135
Real-Variablen im Modus {$N-} 133
Reihenfolge der Belegung 77
Reservierung bei TSR-Programmen 244
Rücksprungadressen in Register 87
selbstunterbrechende Programme 244
Standardgröße 51
statische Unterteilung für Interrupts 473
Umsetzen 239
Umspeicherung 8087 -> 8086# 120
und das BP-Register 52
und Rekursion 193
Wertparameter 63
Standardgröße des Stacks 51
Start des Systems 2
Start- und Endcode
für Funktionen 51
TURBO 3.0 53
für Interrupts 237
nicht bei inline-Deklarationen 84
statische Anwendungen von Zeigern 181
Statuswort-Register des 8087 102
Strings
ASCIIZ 292
Ausgabe 45
TURBO 3.0 47
formatierte Nullstrings 475
Funktionen 56
Ergebnistypen 60
TURBO 3.0 60
und Rekursion 194
Lokalkopien 70
Umcodierung für OutText 400
und GetMem 212
Write 417
Zuweisungen 58
StrNCopy (Routine von System) 58
Struktur
dynamisch geladene Grafiktreiber 321
eines typischen Programms 13
statisch eingebundene Grafiktreiber 325
von CE (Zeichensatz-Editor) 361
von Dateivariablen 410
von Graph 315
von Variablen 164
Teilzugriffe 175
von Zeichensätzen 344
strukturierte Wertparameter 63
Lokalkopie 70
TURBO 3.0 72
SYMDEB, Probleme mit Graph 321
System (Unit)
Initialisierung 22
Coprozessor 105
Interruptvektoren 25
Interruptvektoren, V 5.0 30
Lage 18
Routinennamen 27, 468

T

Tag-Register des 8087 100
Tastaturpuffer, Organisation 307
Tastenbefehle von TPCED 294
Tastendrücke simulieren 307
Teilzugriffe auf Strukturen 175
temporärer Speicherbereich 57
TextAttr 35
Textfenster, Initialisierung 38
TextHeight (Standardfunktion) 348
TextRec (Recordtyp) 410
Redeklaration 413
TextWidth(Standardfunktion) 348
TOS (Top of Stack) 97
TPCED
als TSR-Programm 302
Tastenbefehle 294
transienter Teil von COMMAND.COM 3
Treiber-Routinen für Dateivariablen 411

TSR-Programme 230
aktuelle Entfernung 258
allgemeine Lösung 251
CATCH22 291
Control-Break 306
DOS-Überwachung 262
Eckdaten 245
entfernbare 245
Environment-Tabelle 250
Ermittlung der Eckdaten 257
Ermittlung des PSP 250
Ersatz von DOS-Funktionen 288
funktionsgesteuert 246
»Freigabe« 250
»Selbstunterbrechung« 244
Kennbytes 249
mehrere über einen Vektor 261
Mehrfach-Installation 245
und Vektorketten 262
PRTFILE 271
Prüfung auf »OFF« 256
Reihenfolge 262
TPCED 302
Erweiterungen 309
TRACE287 279
TRACE86R 253
zyklischer Aufruf 235
über DOS 264
TURBO 3.0
Dateivariablen 409
Ersatz von ConInPtr und ConOutPtr 420
Fließkomma-Konstanten 47
Funktionen 53
Konstanten 46
Laufzeitbibliothek 16
lokale Variablen 66
Parameter 66
Programmstruktur 16
Real-Variablen 135
Spezialdateien 39
Start- und Endcode 53
Stringfunktionen 60
Unterschiede Prozeduren/Funktionen 76
Variablenparameter 69
Turbo3 (Unit), Initialisierung 42
TURBO.EXE und KEEP 231
typisierte Konstanten 43
allg. Vergleich mit TURBO 3.0 46
Graph 20
Speicherung 44
TURBO 3.0 135
System in Version 5.0 15
von Units 18
typisierte Zeiger 164
Typ-Umwandlungen 172
implizite in der Version 5.0 125
verschachtelte 470

U

Umdeklarieren der Fließkommatypen 141
umgekehrte polnische Notation 99
Umleitungsmöglichkeiten für Ein- und Ausgabe 416
Umwandlung von ASCIIZ-Strings 292
Unendlichkeiten 101
Units
gemeinsame Segmente 33, 42
Initialisierung 21
Lage im Speicher, Variablen 18
typisierte Konstanten 18
Unterprogramme 429
untypisierte Konstanten 43
allg. Vergleich mit TURBO 3.0 46
Speicherung 44
TURBO 3.0 135
untypisierte Zeiger 168
UpCase, neue Version 86
als inline 88
UpperCase, neue Version 89
Übergabe siehe Parameter
Überläufe, Verhindern durch Typ-Umwandlung 172

V

Variablen
Adressen 164
Version 5.0 20, 164
dynamische 183
Größe 164
Größe in Maschinenprogrammen 449
Struktur 164
Teilzugriffe 175
von Units 18

Variablenparameter 63
  Adressierung 66
  getarnte 92
  im Modus {$N+} 130
  TURBO 3.0 69
Vektor-Adresstabelle in Zeichensätzen 344
Vektordefinitionen von Zeichen 345
  Optimierung 355
Vektorketten
  Ein-/Ausgabe 419
  Exit-Prozeduren 24, 36
  Interrupts 262
  Zeiger auf Routinen 185
Vergleiche
  mit TURBO 3.0 siehe TURBO 3.0
  mit der Version 5.0 siehe Version 5.0
  von Fließkommawerten 103
Version 5.0, Unterschiede 11, 15, 20, 28, 38, 41, 49, 94, 107, 108, 121, 125, 140, 232, 316, 332
Videomodus, Setzen durch Crt 34
Vorsicht mit
  BP-Register 53
  const-Deklarationen und inline 83
  DOS-Aufrufen in TSR-Programmen 263
  dynamisch belegten Strings 212
  indirekten Sprüngen 476
  inline 77
  inline-Symbolen 80
  interrupt-Deklarationen 236
  KEEP und TURBO.EXE 231
  Pseudo-var-Parametern 92
  Readln und ^string 220
  Release und Graph 323
  TSR-Programmen und Control-Break 306
  TURBO-eigenen Vektoren 259
  Typ-Umwandlungen
    bei Arrays 173
    implizite (Version 5.0) 125
    nicht bei Funktionszeigern in der Version 5.0 189
    von Gesamtausdrücken 172
  zuweisungskompatiblen Zeigertypen 167

W
WAIT-Befehle 96
  und 80286/87 469
Wert-Konvertierungen 173
  Fließkomma-, implizite in V 5.0 125
Wertparameter 62
  Adressierung 63
Write 417
  allg. Vergleich mit TURBO 3.0 47
  formatierte Nullstrings 475
  InOut- und Flush-Funktion 417
  ohne DOS-Aufruf 242
  typisierte, untypisierte Konstanten 44

Z
Zeichenbreiten-Tabelle in Zeichensätzen 344
Zeichendefinitionen
  Anhängen und Löschen 354
  grafische Bearbeitung 355
  Optimierung 355
Zeichensätze 343
  Ausgabe mit Pascal 348
  dynamisch geladen 323
  Editor CE 356
  Erweiterung 352
  Funktionscodes 346
  Kopfinformation 343
  Skalierung 347
  Umcodierung für deutsche Umlaute 400
  Vektordefinitionen 345
Zeiger 163
  Arrays mit 181
  auf Pascal-Routinen 185
    Version 5.0 189
  Begriffsdefinition 163
  »freigegebene« 184
  inkompatible 168
  Normalisierung 438
  Notation 168
  OpenFunc..CloseFunc 411
  Typ-Umwandlung 176
    Nachgedanken 470
  und Records mit Zeigern 170
  Zerlegung in Segment und Offset 177
Zeilenzahl des Bildschirms 37
Zeitmessung über BIOS-Ticks 143
Zusammenfassung der Kapitel 467

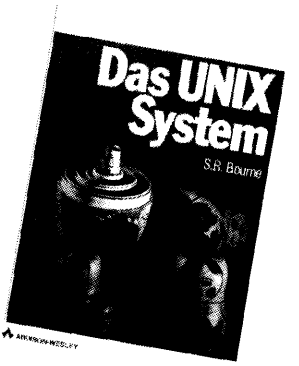
Zuweisungen
  an Teile von Strukturen 176
  Fließkommawerte im
    Modus {$N+} 121
  kompatible Zeigertypen 167
  Real-Variablen im Modus {$N-} 134
  Strings 58
    dynamische 212
  über Zeiger 164
Zwischenspeicherung von
  Funktionsergebnissen 50
  Fließkommawerten 127
  TURBO 3.0 53
zyklischer Aufruf
  ohne zeitliche Überladung 263
  über den »Clock Tick« 235

UNIX

S.R. Bourne

DAS UNIX-SYSTEM

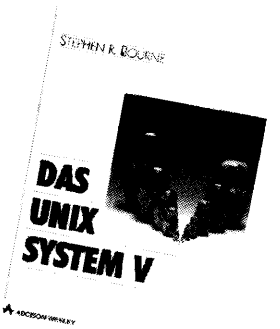
Das Buch beinhaltet eine Einführung in die wichtigsten Komponenten von UNIX, das Datensystem, die Texteditoren und die Kommandosprache „Shell". Weitere Kapitel über die Programmiersprache C und die Dokumentaufbereitung vertiefen das Wissen.
467 Seiten, 1985, ISBN 3-925118-17-9



S.R. Bourne

DAS UNIX-SYSTEM V

Die Fortführung von Bournes „UNIX-System" führt als umfassender und praxisnaher Leitfaden in alle wichtigen Komponenten wie Dateiensysteme, Texteditoren und Kommandointerpreter „Shell" ein und stellt die Programmiersprache C ausführlich dar.
448 Seiten, 1987, ISBN 3-925118-23-3



Keith Haviland/Ben Salama

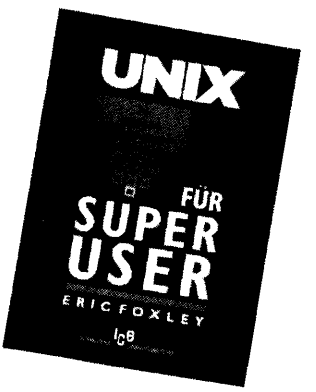
UNIX SYSTEM PROGRAMMING

Ziel dieses Buches ist es, das Programmier-Interface zu beschreiben, das zwischen UNIX und der Anwendersoftware, die unter UNIX läuft, existiert. Die Autoren stellen außerdem eine Menge an Übungen und Beispielen zur Verfügung.
256 Seiten, 1987, ISBN 0-201-12919-1

Eric Foxley

UNIX FÜR SUPER-USER

Für jeden, der bereits mit den Grundprinzipien von UNIX vertraut ist, stellt „UNIX für Super-User" eine unerschöpfliche Informationsquelle dar. Es enthält sowohl allgemeine Prinzipien zur Gewährleistung der Sicherheit und Leistungsfähigkeit des Systems als auch eine Beschreibung aller wichtigen Operationen.
271 Seiten, 1988, ISBN 3-925118-24-1



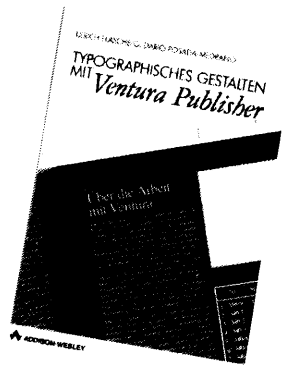
ADDISON-WESLEY VERLAG (Deutschland) GmbH

DESKTOP PUBLISHING

U. Flasche/D. Posada-Medrano

TYPOGRAPHISCHES GESTALTEN MIT VENTURA PUBLISHER

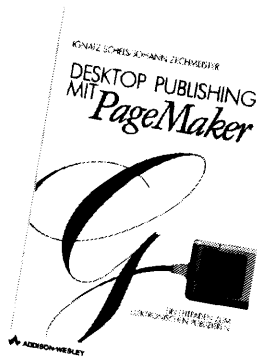
Ventura Publisher ist das leistungsstarke Satzprogramm für MS-DOS-Rechner. Dieses Buch zeigt anhand von Musterbeispielen, wie einfach und effektiv Texte gestaltet werden können.
250 Seiten, 1988, ISBN 3-925118-35-7



I. Schels/J. Zechmeister

DESKTOP PUBLISHING MIT PAGEMAKER

Die Autoren führen anschaulich und gezielt in die Thematik Desktop Publishing ein. Sie vermitteln grundlegende typographische und drucktechnische Kenntnisse sowie die vielfältigen Möglichkeiten von PageMaker.
264 Seiten, 1988, 3-925118-57-8



ADOBE SYSTEMS INC., POSTSCRIPT EINFÜHRUNG UND LEITFADEN

277 Seiten, 1987, ISBN 3-925118-70-5

ADOBE SYSTEMS INC., POSTSCRIPT HANDBUCH

350 Seiten, 1988, ISBN 3-925118-79-9
Beide Bücher wurden von Adobe Systems Inc., den Entwicklern von PostScript, selbst verfaßt. Sie bieten eine grundlegende Einführung in die Sprache und bringen viele Beispiele, die die Möglichkeiten von PostScript demonstrieren.

T. Bove/C. Rhodes/W. Thomas

DIE KUNST DES DESKTOP-PUBLISHING

„Desktop-Publishing" hilft bei der Einrichtung eines elektronischen „Verlagshauses" direkt auf Ihrem Schreibtisch. Mit Ihrem Computersystem und der entsprechenden Software haben Sie jede Möglichkeit, Broschüren, Kataloge, Info-Material, Anzeigen, ja selbst Zeitschriften und Broschüren herzustellen. Dabei entfallen die Umwege über Satzbetriebe und Grafikstudios genauso wie Schere und Klebstoff.
330 Seiten, 1987, ISBN 3-925118-68-3

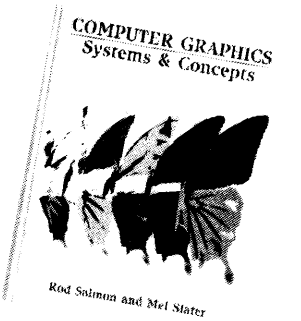
ADDISON-WESLEY VERLAG (Deutschland) GmbH

COMPUTERGRAFIK

R. Salmon/M. Slater

COMPUTER GRAPHICS

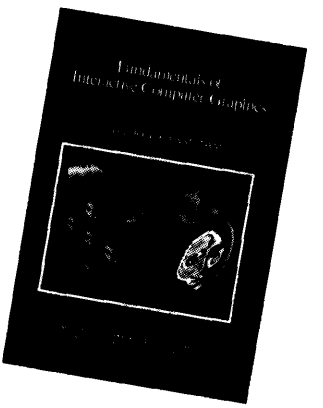
Dieses Buch wurde sowohl für den Anfänger als auch für den fortgeschrittenen Anwender geschrieben, es führt den Leser in weitverbreitete und gebräuchliche Computergrafik-Systeme ein. Weiterhin bietet das Buch dem Leser eine Übersicht über die notwendigen Schritte, um ein Computergrafik-System aufzubauen.
600 Seiten, 1987, ISBN 0-201-14656-8



Neil Weinstock

COMPUTER ANIMATION

Der Autor hat hier ein Buch für den fortgeschrittenen Anwender geschrieben. Der Schwerpunkt liegt dabei auf der Darstellung von passenden Algorithmen für Animations-Sequenzen. Die meisten Beispiele wurden in C geschrieben, der führenden Sprache für den professionellen Anwender.
432 Seiten, 1987, ISBN 0-201-09438-X



J. Foley/A. van Dam

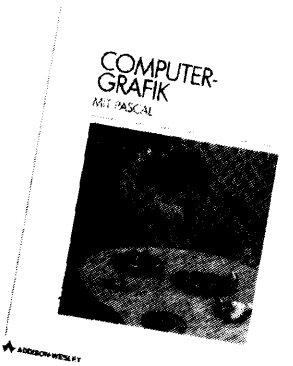
FUNDAMENTALS OF INTERACTIVE COMPUTER GRAPHICS

Hier finden Sie, was Sie schon immer über Computergrafik wissen wollten — und noch einiges mehr. Dies ist wahrscheinlich das umfassendste Buch über die Grundlagen der Computergrafik. Darüber hinaus: Reichhaltige farbige Illustrationen.
664 Seiten, 1982, ISBN 0-201-14468

Marc Berger

COMPUTERGRAFIK MIT PASCAL

Diese detaillierte Einführung bietet eine umfassende Behandlung grundlegender Grafikelemente und wichtige fortgeschrittene Konzepte. Dieses Buch beinhaltet alle Grundbegriffe, fortgeschrittene Themen, Diskussion der Grafikhardware.
400 Seiten, 1988, ISBN 3-925118-74-8



ADDISON-WESLEY VERLAG (Deutschland) GmbH

3893191305
* 1 2 5 3 4 8 *
679
Turbo Pascal 5.0 : [Konzepte, Analysen, Tips und Tricks].

„... ein exzellenter Schreibstil, der eine „spitze Feder" mit vereinigt. Als Gesamteindruck verbleibt, daß man sich sch zweiten Teil freuen kann (...). Dieses Buch kann für ernsthc „Muß" bezeichnet werden."

PASCAL, 5/88

„... für mich der mit Abstand beste Buchkauf seit langem."

Martin Gerdes, c't 11/88

So sah die Reaktion der Fachpresse auf Turbo Pascal 4.0: Konzepte, Analysen, Tips & Tricks aus – und nun hat Arne Schäpers seinen Bestseller um die Hintergründe der neuesten Version von Turbo Pascal erweitert.

In diesem Buch finden Sie:

- *Speicherstrukturen* – Interner Aufbau von Turbo-Programmen (Parameter, Funktionsergebnisse, lokale Variablen etc.) mit einem Vergleich der Versionen 3.0, 4.0 und 5.0, der die Unterschiede bis aufs letzte Bit klärt
- *Programmierung des 8087* – Aufbau, paralleles Arbeiten, Initialisierung und Verwendung durch Turbo Pascal, die Funktionsweise des Emulators der Version 5.0 und der – fast unglaubliche – Zweck der diversen Interrupts. Dazu kommt eine Routine, die eine Apfelmännchen-Gesamtansicht in weniger als 2 Minuten zeichnet
- *Grundlagen* – Zeiger, Rekursion und Listen mit verständlichen Beispielen
- *Speicherresidente Programme,* die sich auch ohne einen Neustart des Systems wieder entfernen lassen und – natürlich – in beiden Versionen des Compilers einsetzbar sind
- *Grafik* – Deutsche Zeichensätze, Struktur von Graph und Grafiktreibern in den Versionen 4.0 und 5.0, 640 x 480 Pixel für GENOA-Karten, Grundlagen für Erweiterungen auf beliebige (!) Bildformate, Laden und Speichern von Bildern unabhängig vom Grafikmodus
- *Gerätetreiber* – Schnellere Druckerausgabe, Simulation der 3.0-Zeiger für Ein- und Ausgabe

Dazu kommen einige Programme, die auch auf Diskette erhältlich sind:

- DATEFIND sucht Festplatten über Datum und Uhrzeit ab
- TRACE 287 schrittweises Verfolgen von Coprozessor-Befehlen mit DEBUG
- TPCED ein Kommandozeilen-Editor in Turbo Pascal
- CE ein Editor für Grafik-Zeichensätze
- FASTLIST beschleunigt Drucker-Ausgaben um das Dreifache

ISBN 3-89319-130-5
VVA-Nr. 563-00130-9

9 783893 191307

ADDISON-WESLEY VERLAG (DEUTSCHLAND) GmbH